An Abstract of the Thesis of

Nanyu Cao for the degree of Master Science in Computer Science presented on June 20, 2000.
Title: Temporal Programming in Grid-Oriented Visual Programming Languages

Abstract approved: ___
Margaret M. Burnett

Specifying varying speeds and temporal relationships is necessary when programming graphical animations, but support for temporal programming has usually been done by adding new language features to a Visual Programming Language (VPL), and these features must be mastered over and above the other aspects of the VPL. However, some researchers have believed that time should be able to be treated like just another dimension. In this thesis, we explore whether temporal programming can indeed be done using exactly the same devices as in spatial programming in grid-oriented VPLs. Toward this end, we provide a continuum of models aimed at this goal and discuss their advantages and disadvantages. Also, we identify core issues that help illuminate the essence of the problem.

Temporal Programming in Grid-Oriented Visual Programming Languages

by

Nanyu Cao

A Thesis Submitted

to

Oregon State University

In partial fulfillment of

the requirements for the

degree of

Master of Science

Completed June 20, 2000

Commencement June 2001

Master of Science thesis of Nanyu Cao presented on June 20, 2000.

Approved:

Major Professor, representing Computer Science

Head of Department of Computer Science

Redacted for privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for privacy

Nanyu Cao, Author

Acknowledgment

I am grateful to my major professor, Dr. Burnett of the important ideas, useful hints, precious comments she puts in my work. This thesis was extensively discussed in a series of meetings with her. John Atwood was also directly involved. Thanks for his contributions in this thesis. My appreciation goes to Dr. Rothemel, Dr. Quinn and Dr. Harter for their help and being my committee. Thanks also go to all Forms/3 group members who gave me their valuable suggestions.

Table of Contents

Table of Contents (Continued)

List of Figures

List of Figures (Continued)

List of Tables

Temporal Programming in Grid-Oriented Visual Programming Languages

Chapter 1: Introduction

Many programming languages do not provide explicit control to the programmer over the temporal aspects of a program. Instead, in most languages, a program's temporal behavior, such as the relative rates of progress between two concurrent tasks, are side effects of the program's execution.

However, there are important applications in which temporal behavior is critical to a program's correctness, especially in applications involving visual data. One example is animated graphics, a popular domain for visual programming languages (VPLs). Since many visual programming languages (VPLs) offer strong support for visually programming with visual data, (e.g., Cocoa [13] now known as StageCast, Altaira [24], Visual AgentTalk [16], ToonTalk [17], user interface programming in Prograph [25], EUPHORIA [21], and Forms/3 [3,4,6]) it is important to consider ways to support visually programming temporal behavior. Temporal programming is especially relevant in multimedia applications and in programming dynamic graphics and animations. Another motivation lies in software visualization.

In earlier work in Forms/3 [2,3,6], the Forms/3 group showed how adding a time dimension to the spreadsheet paradigm facilitates programming of animated graphics. Our interest was in the realm of software animation, and they showed a straightforward way to program an animation such as the one in Figure 1 [6]. However, another goal of algorithm animation is to program such an animation without modifying the original (sorting) algorithm. Although they had initially believed taking this step would be straightforward, they later discovered this not to be the case. In particular, as Figure 2 demonstrates, two problems arose in programming the temporal relationships between the sort and its animation:

Problem 1: How to program temporal behaviors in a notation closer to how they evolve over time than the one-dimensional syntax of `earlier` and `fby`.

Problem 2: How to program temporal interrelationships without artificially inserting code such as `counter`, whose purpose is to slow down the algorithm.

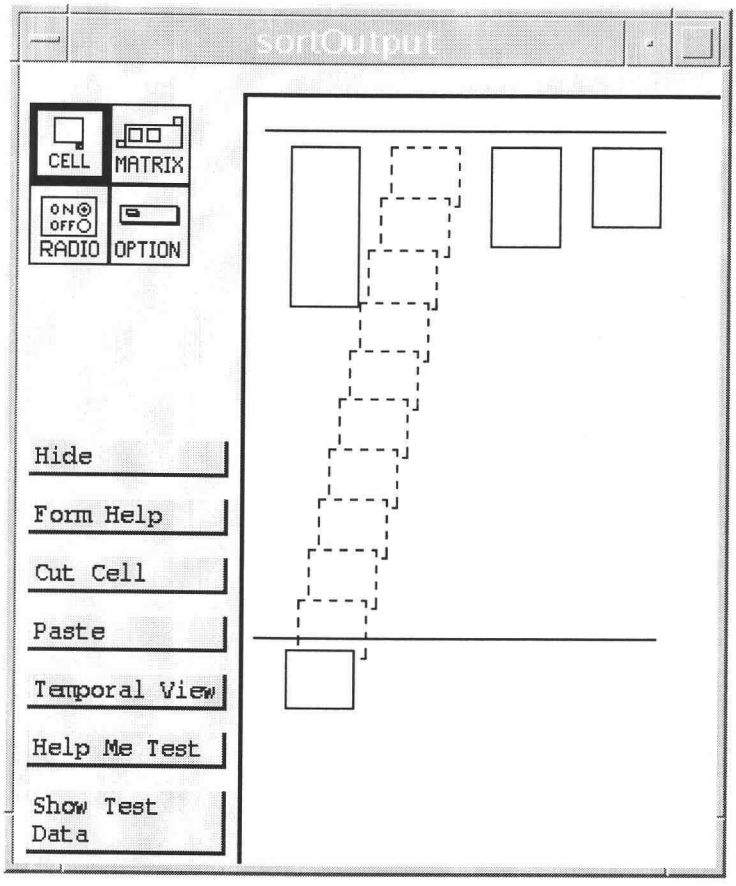


Figure 1: The animation of a sorting algorithm in Forms/3. A moving box drops down gradually (10 steps) into place when its corresponding element is inserted into a sorted group. This screenshot has been annotated with the dotted entries to show moment over time.

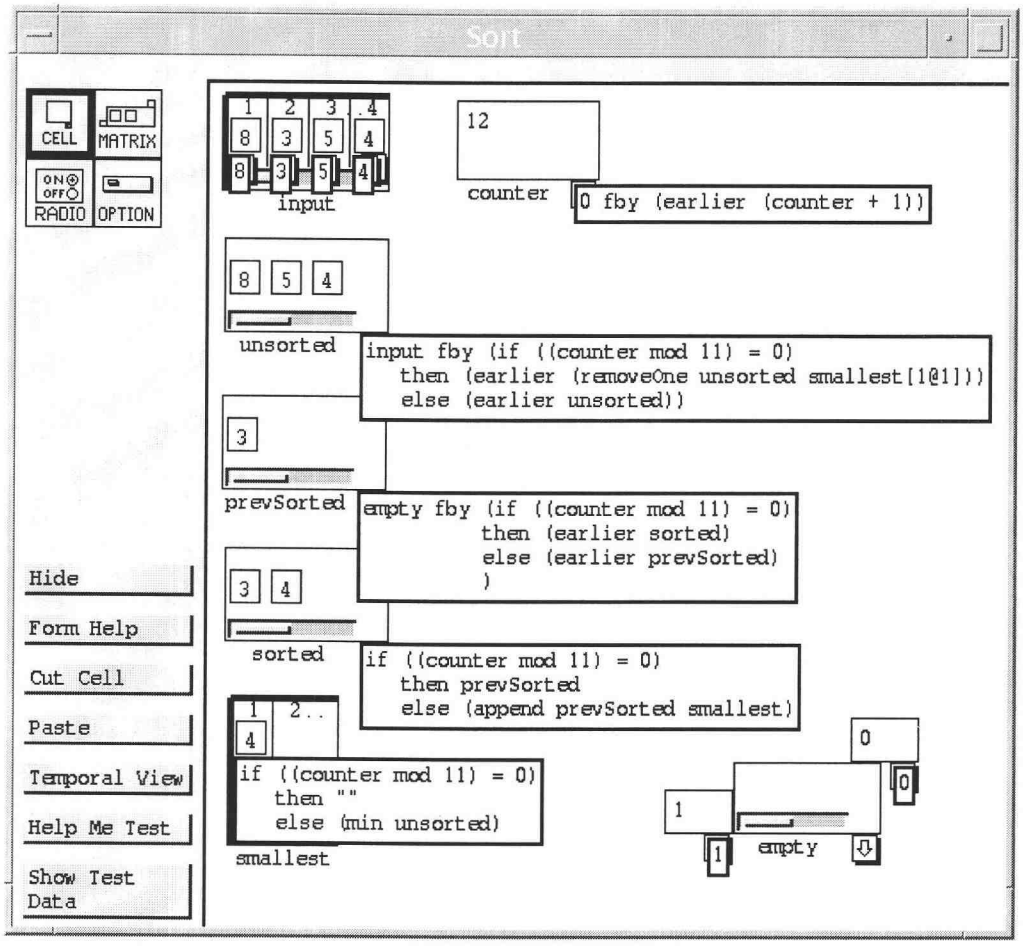


Figure 2: MatrixSort, as implemented in Forms/3 prior to this thesis

In this thesis, we explore the problem of whether visually programming in time can be done via exactly the same techniques used in programming in space. We will focus on ways to programmatically control speed and temporal interrelationships among elements of the program, such as are needed in programming animated graphics, but will ignore the issue of programming temporal deadlines. (The latter is important in some domains, but is not required for most animated graphics programming.)

We will consider this problem in grid-oriented VPLs. These include members of the spreadsheet paradigm, as well as other grid-oriented VPLs. The problem is particularly relevant to VPLs of this group supporting grid-based programming of graphics, such as AgentSheets [16], Cocoa [13], Forms/3, NoPump [31], Spreadsheet for Information Visualization [6], and Spreadsheet for Images [19].

The contributions of this thesis are two. First, several models that solve variants of this problem are presented. Second, differences between temporal and spatial programming are identified that can cause difficulties in supporting grid-based programming of temporal behavior, because they can lead to language design problems that are known to interfere with humans' programming abilities.

In Chapter Two, after the discussion of related work, we give some background about cognitive dimensions, spatial programming and temporal programming in Forms/3. The sort algorithm animation example is also described. In Chapter Three, we propose the continuum of new time models. In Chapter Four, these time models are explained in detail with examples. Via the evaluations of these time models by cognitive dimensions and other language design issues, their advantages and disadvantages are discussed. Implementation issues are the topic of Chapter Five. The conclusions and future work are given in Chapter Six.

Chapter 2: Background

2.1 Related work

Although there are several VPLs that support *visualization* of temporal behavior (e.g., [8,11]), many VPLs have not explicitly supported *programming* of temporal behavior.

Some early VL work regarding temporal aspects of software was done to support temporal logic in program specifications. The earliest such VL approach is probably GIL (Graphical Interval Logic) [18]. GIL is a visual temporal logic for reasoning about the temporal properties of programs. It is a high-level, proof-oriented approach aimed primarily at visual specification and subsequent comparison to a (separate) program's properties. While there is also a tool supporting generation of a model from GIL temporal specifications, it does not support programming other than of temporal specifications.

There have been numerous examples of VPL research about coding temporal information via devices specific to temporal programming. Duisberg's work with temporal constraints [9] and with use of music notation devices [10] are perhaps the earliest VPL work aimed at temporal characteristics of animated graphics. Later examples include programmatic use of diagrams oriented toward time, such as space-time diagrams and concurrency maps as in [27] and time lines as in [14].

Treating time as temporal events has been another approach. For example, in Pavlov [32], there is an integer global time. Programmers explicitly specify what will happen at time n or every n times. However, it is not possible to refer to other objects' previous states. Pavlov's temporal programming is unlike space programming because unlike the other aspects of Pavlov, temporal information is not inferred from demonstrations. Visual Basic is another example of a temporal model similar to Pavlov's, which fits well in Visual Basic because of its reliance on the event model for other GUI aspects as well.

There has also been significant work on temporal aspects of programming multimedia applications. Like the work described above, this body of work also provides constructs specific to temporal programming. One of the earliest examples of such work in VPLs was

TYRO [20], a demonstrational, constraint-oriented multimedia authoring language. TYRO is intended as a "designer's apprentice," and infers the correct temporal layouts from a combination of constraints and examples programmed using temporal-specific mechanisms. More recently, Song et al. developed an elastic time model, in which the deadline constraints are specified as the endpoints of a "spring", and the system uses these as constraints to schedule the computations that need to fit in this spring [26]. Their time-box representation provides a visual way of specifying these spring endpoints and their relationships to other springs.

Approaches such as the above support temporal programming by introducing specific models and subsystems that relate only to time. The other possibility is that time is just another axis in a coordinate system, i.e., a t-axis in a coordinate system already containing an x-, y-, and perhaps a z-axis. This approach was used in an early spreadsheet language based on Lucid [29] known as Plane Lucid [7], which gave 2-D operators such as hsby and vsby for "horizontally/vertically spatially by (to the right/below)" to reference elements along the x- and y-axis of a spreadsheet grid and analogous operators such as `fby` meaning "followed by (temporally next)" to reference elements along the t-axis. The earlier Forms/3 approach discussed in the introduction was influenced by this work.

Gamut [22] is a programming-by-demonstration VPL that also supports some temporal programming without introducing new models. The form of temporal programming it supports is referencing the previous state of an object. This is done by inclusion of faded "temporal ghosts" in the demonstration area, so that they can be included in the demonstrated logic. The approach corresponds to our design goal of allowing a space-oriented programming technique to be applied to time, but does not allow time-oriented logic such as relative speed to be included, and allows reference only to the just-previous state, not to any other previous moments in time.

To summarize, previous VPL research that relates to temporal behavior can be divided into two categories. One is to provide entirely new language devices designed explicitly to support programming in time. The other is that time is just another dimension, able to be handled exactly like other dimensions. Our work falls in the latter category.

2.2 Cognitive dimensions

It is possible to bring research into cognitive issues of programming to bear upon VPL design decisions by considering Green's and Petre's *Cognitive Dimensions*, a distillation of psychology of programming knowledge into a form usable by non-psychologists [12]. This work has motivated some of our design goals. For example, solving Problem 1 would improve *Closeness of Mapping*, the similarity of the programming notation to the problem being solved (also termed *directness* in the HCI community [15]).

Consistency is another of the cognitive dimensions particularly pertinent to this work: "When some of the language has been learnt, how much of the rest can be inferred?" [12]. Since (spatial) grid-based programming has been shown by spreadsheets' popularity to be easy enough for end users to master, a level of consistency allowing those users to reapply the same techniques to time suggests that users would be able to master temporal programming as well, and this is what motivated our interest in this strategy. Doing so would mean a VPL user would not have to climb a separate learning curve in order to deal with time; rather, the user could simply reapply what he or she already does when programming spatially. Avoiding such learning curves is particularly important when the intended users of a VPL are end-user programmers. In this thesis, we will use the cognitive dimensions as an early evaluation mechanism for design alternatives, primarily to raise warnings when a design decision is potentially at odds with research from cognitive principles of programming.

In this thesis, we have used these cognitive dimensions to evaluate our language design approaches:

- *Closeness of mapping*

The closer the mapping between the problem world and the program world, the easier the problem solving ought to be. Increasing the closeness of mapping is the goal of this thesis.

- *Consistency*

"When some of the language has been learnt, how much of the rest can be inferred?" [12]. Making programming in time more consistent with programming in space in grid-oriented VPLs is the goal of the design approaches of this thesis.

- *Premature commitment*

Sometimes the user needs to make decisions when there is not enough information. Our goal is to prevent this from arising when programming in time.

- *Viscosity*

"Viscosity" is the name of fluid's resistance to local change. Here it means how much work the user needs to put in to effect a small change. In this thesis, we strive to minimize viscosity.

- *Hidden dependencies*

This term describes relations between two components that are not fully visible. We want to reduce the hidden dependencies in our programming language.

- *Visibility*

Also as known as "explicitness", visibility measures how many steps are needed to make a required material visible. VPLs have more advantages over traditional languages in this dimension, and our design goal is to keep this.

When designing new features in programming languages, it can be difficult for computer scientists to see their designs from the point of view of non-experienced end users. The cognitive dimensions framework provides a useful technique for language designers to evaluate design approaches in the early stages. At the same time designers consider ways to conform to each dimension, they must also pay attention to the fact that there are trade-offs. Fixing problems in one dimension may cause problems in other dimensions, as will be seen in the discussion of the different time models in this thesis.

2.3 Programming in grid-space of Forms3

In order to reapply spatial programming devices to temporal programming, we first describe the programming devices in grid-space programming. Here is how grid-based programming works in space. In the spreadsheet system Excel, the user puts a formula in one cell, and then replicates it (copies with the references automatically changed) across as much of the grid as desired. In the spreadsheet system Lotus, the user instead can group several cells in a grid and instruct the system that these cells all share the same formula. (This is a simplified version of an approach pioneered by Viehstaedt and Ambler [28] and extended in

Formulate [1,30].) Forms/3 supports the latter strategy. The cells' formulas can refer to other cells by row and column number indices.

Our work was prototyped using a grid called a *matrix* in Forms/3. To define values for a Forms/3 grid's (matrix's) cells, the user statically partitions the grid into rectangular regions and, for each region, enters a single formula for all cells in it. Each grid has two additional cells, its row dimension cell and column dimension cell, to specify its number of rows and columns. For example, considering Figure 3, m is a matrix with 2 rows and 3 columns.

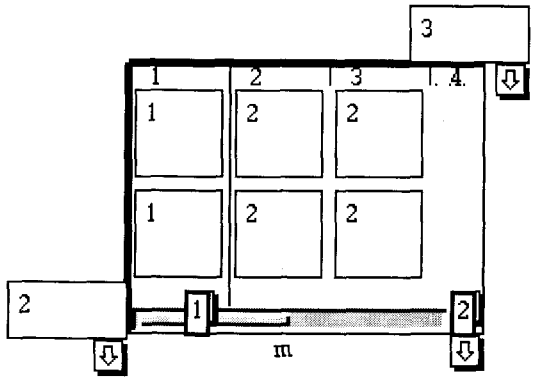


Figure 3: A matrix example

Initially the matrix has only one default region. The user can create new regions by splitting old regions. Dragging the left thick border rightwards can split the original region vertically as in the Figure 3, and dragging the top thick border downwards can split the original region horizontally. In Figure 3, matrix m is separated into two regions. The single line separates the two regions and the formula tab on the right bottom corner of each region shows the formula of the region. All of the cells in this region share this formula. Selecting any cell in this region allows the user to edit the formula by a "formula edit window" which is shown in Figure 4.

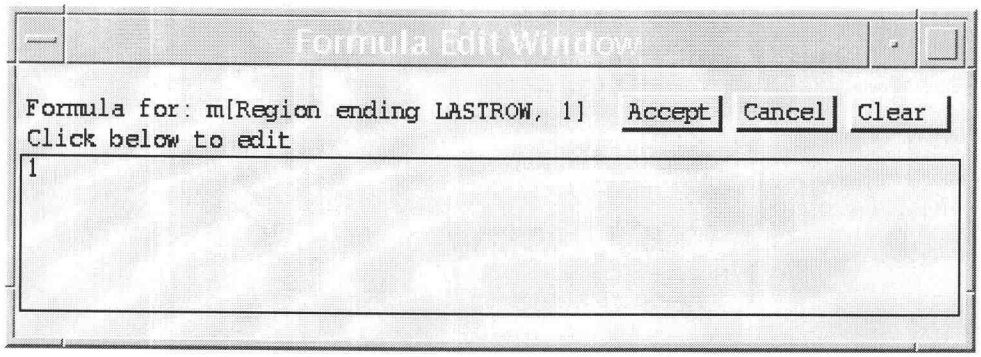


Figure 4: Formula edit window in Forms/3

To statically derive a cell's formula from its shared region formula, any "pseudo-constants" I and J in the formula are replaced by the cell's actual row and column number. For example, in Figure 5, each cell in the right region of matrix m references the cell on its left by "`m[I@J-1] + 1`". This is a commonly used technique of grid-oriented languages such as Excel.

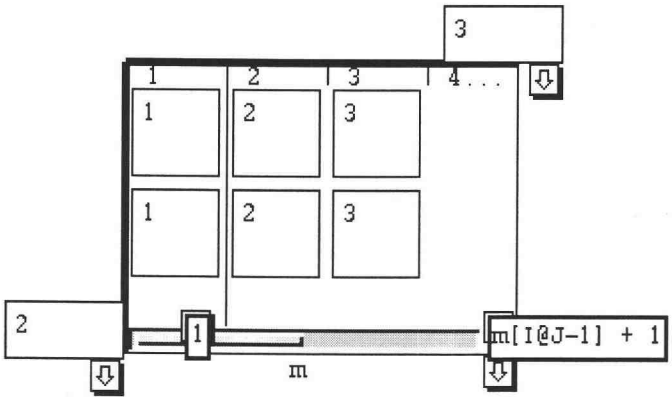


Figure 5: Using I@J notations in a matrix

Forms/3 not only supports region formulas, but also supports a formula for an entire matrix. In Figure 6, matrix `n` refers to the entire matrix `m`.

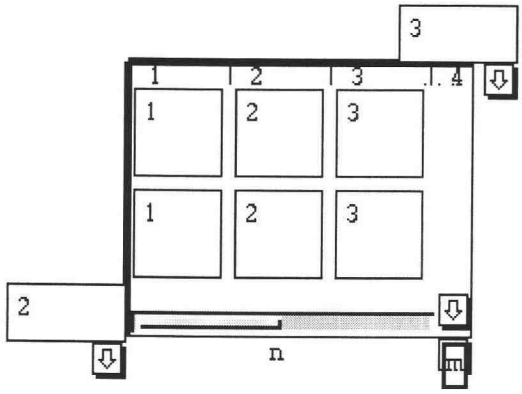


Figure 6: A formula for the whole matrix

Forms3 also supports some matrix operators, most of which were shown in Figure 2.

- `removeOne` matrix value : Removes the first element from the matrix which has the same value.
- `append` matrix1 matrix2 : Appends matrix2 to matrix1. Matrix1 and matrix2 must have the same number of rows.
- `min` matrix : Returns matrix's minimum value.
- `max` matrix : Returns matrix's maximum value.

2.4 Programming in graphics-space in Forms3

Another common programming model of space is computer graphics programming model. In this model, graphics can use different coordinate systems. Three classic operations are translate (move the position of the graphics), scale (scale the size of the graphics) and rotate (rotate the angle of the graphics). These operations allow translating, scaling or rotating a graphic from its local coordinate system to other local or global coordinate systems.

In this thesis, we consider not only ideas from programming in grid-space, but also ideas from programming in graphics-space. Following is the description of graphics-spatial programming in Forms/3.

Using simple graphics operators such as `box`, `circle` and `line`, it is possible to make graphical shapes in cells. Forms/3 also has a "compose" operator to combine them and translate them to new coordinates.

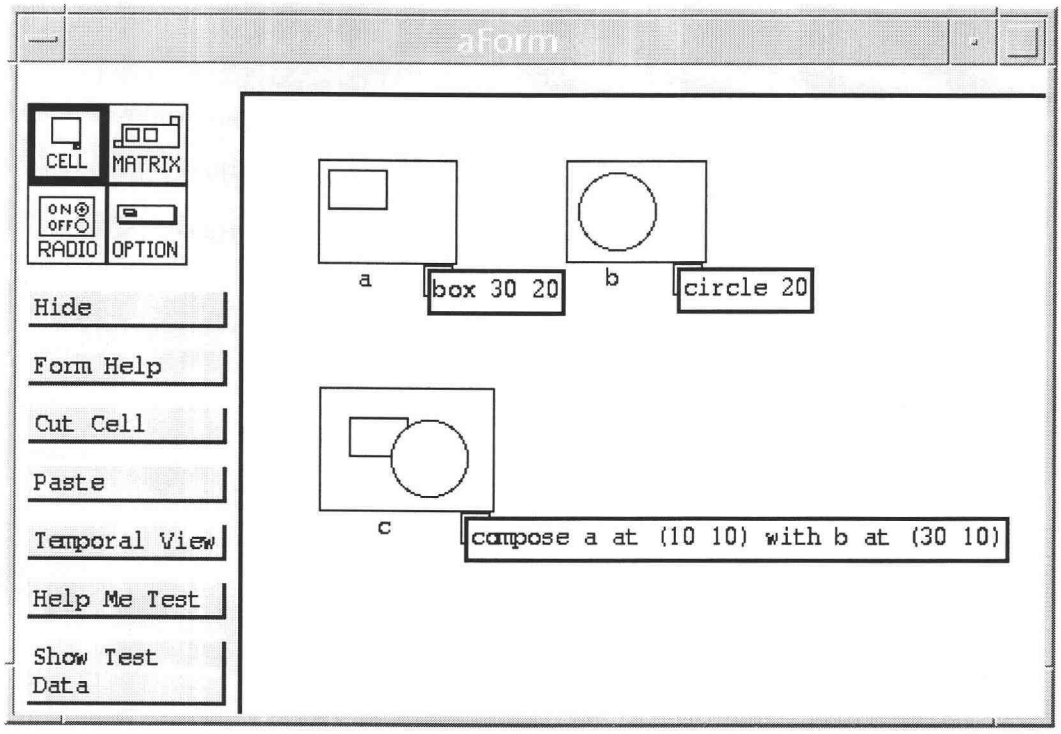


Figure 7: Compose two graphics together

In Figure 7, cell `a` and cell `b` each have their own notion of the origin (upper left of each cell). So does cell `c`, and it combines both cell `a` and cell `b`'s graphics relating to its own origin. By specifying the position with "`at`", the user completes the translate operation.

2.5 Temporal vectors

The formula for a cell does not define just a single value, but rather defines a vector of values along a time dimension. This vector of values is called a "temporal vector". The time

dimension in Forms3 is logical time, ranging from 1 to positive infinity. For example, suppose cell a's formula is "`1 fby ((earlier a) + 1)`". (This follows the Forms/3 temporal operator approach existent prior to the thesis.) Here "`fby`" means "`followed by`". All the value entries of cell `a` after logical time 1 are specified by the expression after the "`fby`". "`earlier a`" means the value of cell `a` at the previous logical time tick. The temporal vector of cell `a` is shown in Figure 8.

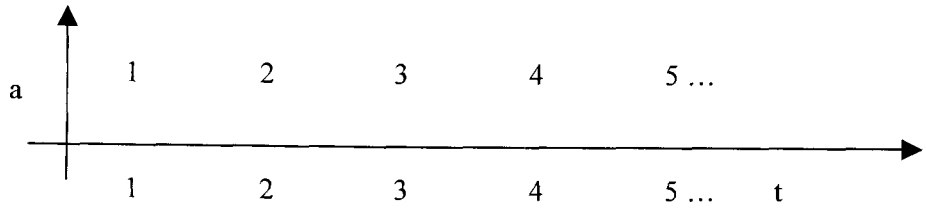


Figure 8: Temporal vector of cell `a`

Every value entry in this temporal vector is called a temporal component and each temporal component has a define time and expire time. For example, the second temporal component's value in Figure 8 is 2, its "define time" is 2, and its "expire time" is 3.

The temporal vector is stored with the cell. In the Forms/3 version existent prior to this thesis, this temporal vector was an interior data structure, and could not be seen by users directly. Users must browse through the time line to see all the values of a cell's temporal vector (one temporal component at a time).

2.6 The animated sort example - MatrixSort

Software visualization is an important application of animated graphics, and thus we have chosen the animated sort as a motivating example for this thesis. In fact, the sort makes a particularly good example because its goal of sorting an ordered group of elements gives it a spatial component to accompany the temporal issue of animating it. MatrixSort is a software

visualization example implemented in Forms/3. We will be referring to this example throughout the thesis.

As shown in Figure 2, MatrixSort implemented an algorithm to sort integer elements in matrix `input`. Figure 2 shows how MatrixSort was implemented prior to the language design changes in this thesis. In Figure 2, there are 6 matrices plus 1 standalone cell (`counter`). The overall logic is that `unsorted` starts as `input`, ("`input fby...`"), and then is either the result of removing the minimum element from its own earlier value or else is just a reference to its own earlier value. `Sorted`'s logic is the mirror image of `unsorted`'s: it is either its own earlier value or else is the result of appending the minimum `unsorted` element to its own earlier value. The minimum element of matrix `input` is referenced by the first subcell of the matrix `smallest`. Matrix `empty` is the initial value of matrix `unsorted`. Matrix `prevSorted` is present to break up what would otherwise have been a long, nested formula for grid `sorted`.

In the animation of the MatrixSort algorithm (shown in the Figure 1), the box's length represents the integer being moved. Whenever an integer element is inserted in the matrix `sorted`, the corresponding box will drop gradually from the top into the right place in the bottom within 10 steps. Animation is visually programmed in Forms/3 by specifying the animation parameters on a copy of the Animation Form. Figure 9 shows the Animation Form for the animation of one sorted element. By specifying the box related to the value of the sorted element to the object cell, and giving the start position and end position, the user specifies how to render the animation of one element in the `animation` cell. The Animation Form then automatically generalizes this specification to the other elements being sorted. Finally, the `Sortouput` form refers to these elements to show the collected animation.

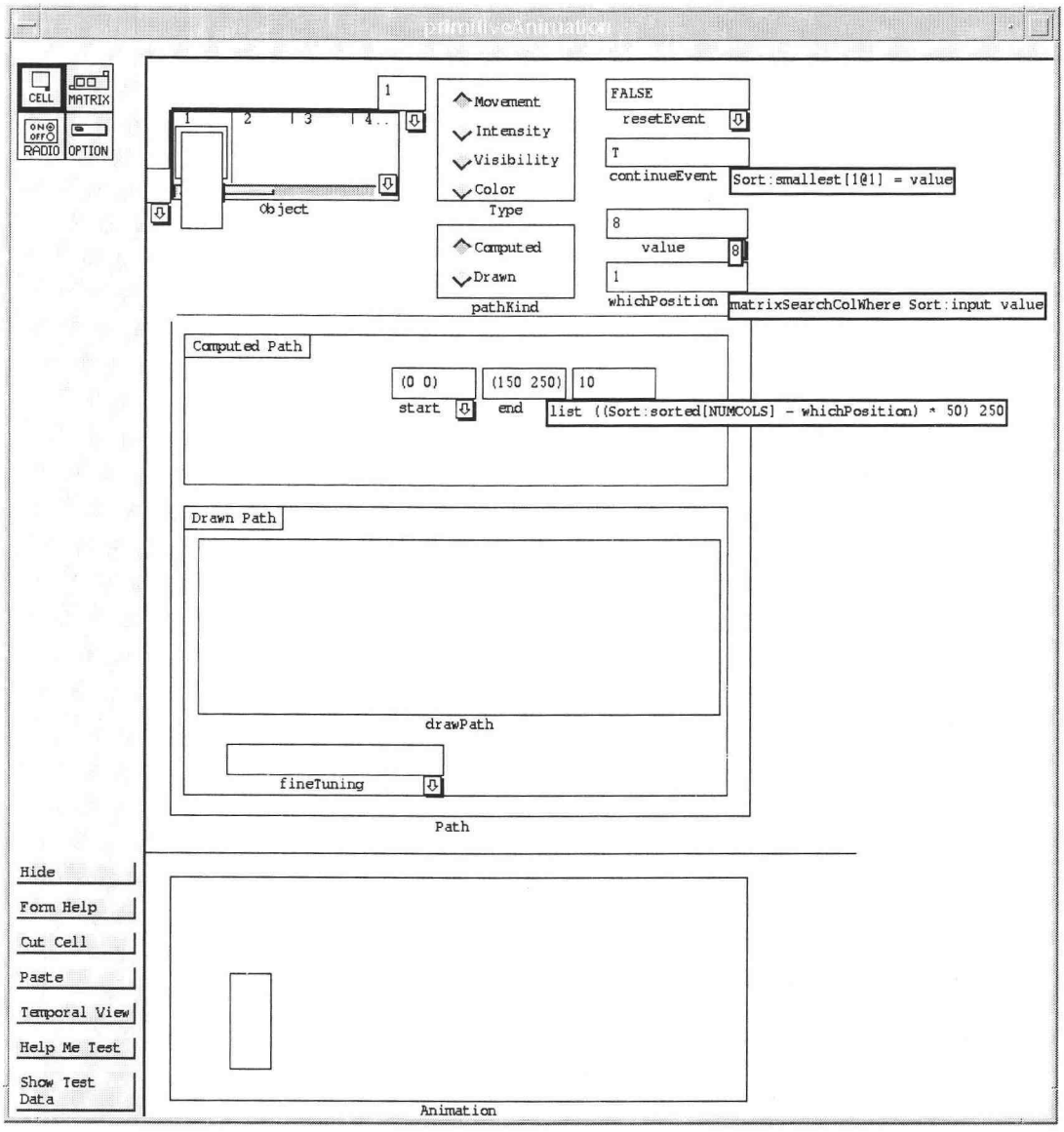


Figure 9: Primitive animation form

Chapter 3: A Continuum of Time Models

"Programming" (defining formulas for) spreadsheet grids has been mastered by thousands of end users. This demonstrated ease of use motivates us to reapply this grid-based way of programming to time.

How grid-based programming works in space is described in Section 2.3. In Section 2.6 we described the MatrixSort animation example in Forms/3. Although matrices in Forms/3 have been empirically shown to be simpler for humans to use for some tasks than are more traditional methods [23], they do not seem particularly simple in Figure 2. Much logic in this program is required for "slowing" down the program via cell `counter` so that the accompanying animation can perform smooth graphical transitions between element insertions into grid `sorted`. To accomplish this slow-down, most formulas contain `if`s that make decisions depending upon the status of `counter`. This is the kind of workaround necessary in languages that do not directly support the ability to program temporal behaviors.

What we would like to do instead is to allow the specification of this temporal constraint without the use of such workarounds. Toward this end, we next consider several models of time to directly support temporal programming.

Several grid-based models of time are presented here in increasing order of power (functionality) but also of expected programming expertise. Thus, a VPL's designer could choose which of these models to investigate by considering not only the power/functionality needed, but also the programming expertise expected of that VPL's intended audience. Given some model's placement along this continuum, we shall evaluate it by the number of problematic VPL design issues it introduces. The issues we consider are cognitive issues raised by Green/Petre's cognitive dimensions [12], and programming language design problems such as limitations and referential transparency. The models are:

1. Base model:

All cells have the same speed. This is the original time model in Forms/3 as it stood before the work of this thesis.

2. SNF model:

Slow, Normal and Fast. A user can choose one of a fixed number of speeds for a cell, (e.g., `cellA` goes Fast). In our instantiation, the fixed number is 3: Slow, Normal, and Fast.

3. k*N model:

A user can specify a cell's speed as being an arbitrary integer constant k times faster/slower than Normal (e.g., `cellA` goes 2 times faster than Normal).

4. k*x model:

A user can specify a cell's speed as an arbitrary integer constant k faster/slower than the speed of another cell x (e.g., `cellA` goes 2 times faster than `cellB`).

Chapter 4: Programming Along the T-Axis

In all models and their revisions, the time dimension is logical time numbered with positive integers.

4.1 The base model

4.1.1 Description

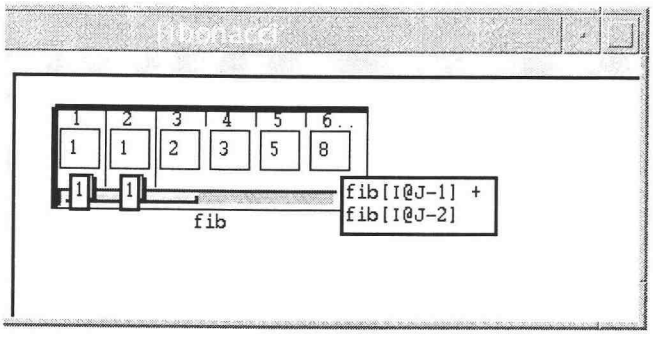
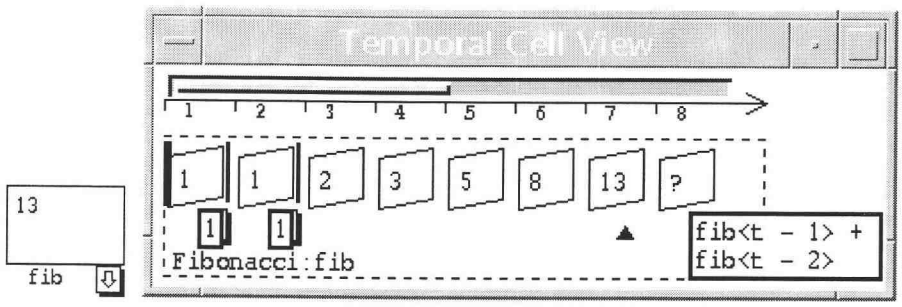


Figure 10: Fibonacci matrix example

Figure 10 shows a (spatial) grid-based approach to programming the Fibonacci sequence, using grid-oriented spatial relationships among the columns. The user separated this grid into the 3 regions shown by dragging the thick border line from the left of the grid rightwards.



(a) (b)

Figure 11: (a) Cell Fibonacci (b) The temporal view of cell Fibonacci

Figure 11 (a) shows a single Forms/3 cell that can compute the Fibonacci sequence temporally under the base model. The computations are specified over time rather than space. The window in Figure 11 (b) shows its 3 regions over time. This window is called the temporal view in Forms/3. In this temporal view, users not only can see the values along the t-Axis, but also can reapply the grid-based way of programming to time. Note the parallels to Figure 10's spatial approach. The small triangle indicates the current time and the "?" marks the next value to be computed.

In Figure 11, the same technique was used to program a Fibonacci sequence over time instead of space, i.e., dragging the region separator border along the t-axis positions instead of the x-axis positions. As with grid-based programming, a formula can reference a cell's value using indices along the axis. In grid-based programming in Forms/3, x- and y-axis referencing is done via format `name[row@col]`; hence temporal referencing is done via format `name<time>`. `t` is a distinguished time meaning "now". For example, in Figure 11, `fib<t-1>` references `fib`'s value at the temporal position just before (to the left on the t-axis).

4.1.2 Evaluation

The temporal view programming device eliminates the needs for the "fby" and "earlier" notations. If a user thinks about the problem as producing a temporal sequence of

Fibonacci numbers, the cognitive dimensions closeness of mapping between the problem and the solution (shown in Figure 11) is closer than with the "fby" and "earlier" notations. This solves Problem 1, because this new programming device – temporal view to program temporal behavior – replaces the one-dimensional syntax of "earlier" and "fby".

In fact, the temporal view is computationally more powerful than "earlier" and "fby", because the latter can not compute the Fibonacci numbers at time 2. The formula after "fby" is effective for all the value entries along the logical time axis other than 1, and thus no formula particular to time 2 is allowed. But with the temporal view, the user can program regions' formulas for any moments in time requiring special processing.

Recall the cognitive dimension of consistency: "When some of the language has been learnt, how much of the rest can be inferred?"[12]. The base model's temporal programming device is consistent with spatial programming in Forms/3. Users can reapply their programming techniques for matrices to temporal programming, such as creating a region and giving a formula to the region. In fact, the *only* thing this model does is add a new axis to the existing grid-based coordinate system. Every cell has exactly the same "logical" speed, i.e. is indexed along the t-axis in the same way as would be done on an x- or y-axis.

Visibility in cognitive dimensions measures the steps needed to make given items visible. Obviously, viewing two related components simultaneously is important for visibility. Before the temporal view was added to Forms/3, the user could only view a cell's values over the t-axis one at a time. Now the user can see all values simultaneously in the temporal view. The relationships between formulas, ensuing values, and the t-axis are also explicit in the way the user enters the formula (by directly moving borders among the values on the t-axis and clicking on other values to reference them) and in the fact that both the formula's temporal aspects and values are explicit in their placement along the time line. For example, the relationship between the first two values and the third one in the Fibonacci example is visible, abstractly in the formulas' placements on the t-axis, and concretely in the values' placements on the t-axis.

Because every cell has the same speed, this model does not have the functionality to solve Problem 2, but it provides a basis upon which the later models can build.

	Problem 1	Problem 2	Closeness of mapping	Consistency	Visibility
Base model	Solved	Unsolved	Improved over Forms/3 prior to this thesis	Improved over Forms/3 prior to this thesis	Improved over Forms/3 prior to this thesis

Table 1: Summary for the base model

Still, even this simple model brings out an intriguing issue about the difference between programming time and space. In space, it is usual for any grid item to be able to reference any other grid item, provided that no cycles are induced. But in time, the ability for the "past" to reference the "future" runs counter to the real world. We believe this flexibility could actually be used to advantage by an imaginative programmer, and would not be encountered by others. However, this is only our own opinion. Empirical work would be needed to solidly determine the extent to which it actually caused problems for users.

4.2 SNF model: Slow, Normal, and Fast

4.2.1 Description

The base model does not solve Problem 2, because all the cells have the same temporal speed. The goal of the SNF model is to solve Problem 2, for at least some cases, in a way usable by users who know only grid-based programming. In order to solve it, this thesis adds support for programming temporal interrelationships.

In keeping with this goal, the SNF model adds to the base model by simply allowing the user to specify one of k speeds for a cell, where k is fixed in the language implementation. In our prototype, k=3, which allows speeds of Slow, Normal, and Fast. Also fixed in the language implementation are the relationships between Slow, Normal, and Fast. (We are

considering allowing the user to modify this via a system preferences dialog.) In this discussion, we will set the speed difference factor to 2.

Using the mechanism in Figure 12, the user can set the speeds of the cells. With this model, in the sorting example, the sorting algorithm's cells could have Normal speed (the default) and the animation's cells could have Fast speed with the difference factor set to 10 instead of 2 as in this figure.

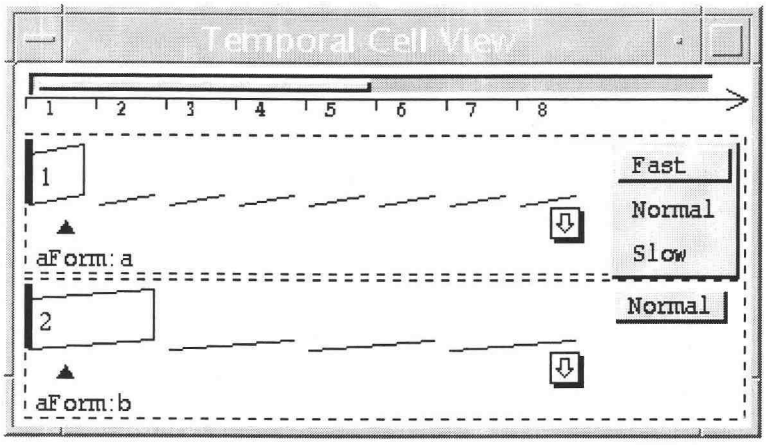


Figure 12: In the SNF model, a user specifies each cell's speed, supported here with a pop-up menu with "Fast, Normal and Slow".

Allowing these multiple rates of speed raises the following question: what do temporal references such as Figure 11's <t-1> mean under this model? This question can be answered via the following three semantic rules:

Rule 1. The t-axis: As in the base model, in the SNF model there is one and only one time axis, it is known globally, and its indices are 1,2,... (These characteristics are the traditional characteristics of x- and y-axes.) We term each position along the t-axis a *tick*.

Ticks are not tied to "real" time units such as seconds or minutes; they are simply progressive positions on the t-axis. In Figure 13, the ticks are represented by the short vertical lines among the numbers on the t-axis. Fast speed allows values at every tick (1,2,3...), provided that dependencies in the formulas deliver enough data to populate every tick, Normal speed allows values at every 2 ticks, (1,3,5...), and Slow speed allows values at every 4 ticks

(1,5,9...). For example, in Figure 13, cell a with Fast speed has value 10,20,30 at time 1, 2, 3..., cell b with Normal speed has value 10,30,50... at time 1,3,5..., and so on.

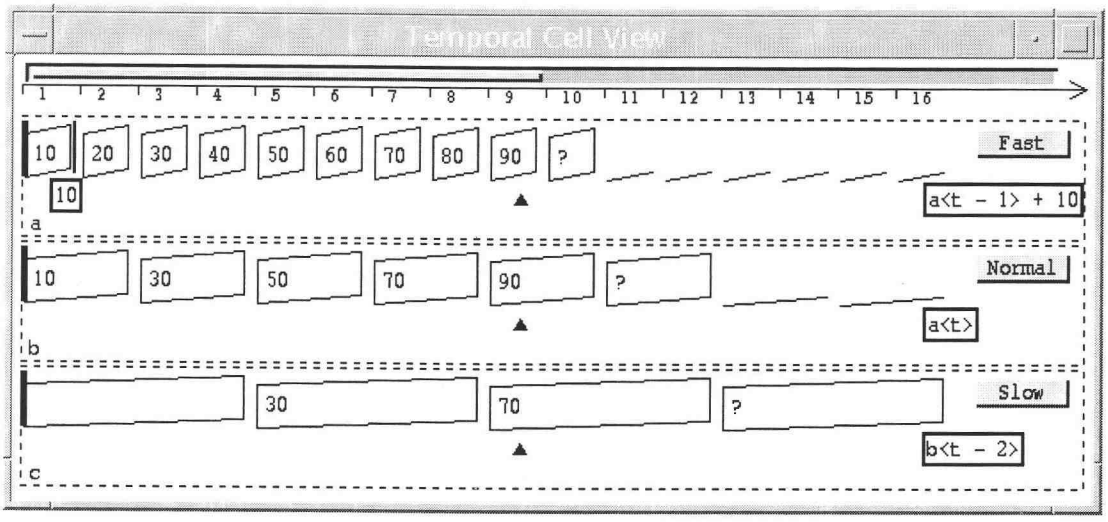


Figure 13: In the SNF model, cell values are located along a global t-axis as shown.

In this and all the models in this thesis, if there is a need to compute with or display a value at a time not explicitly populated for that cell, the most recent value before that time is still considered to be current. For example, if a slow cell's value at time 3 is needed, its value at time 1 is considered still current and is used. For example, in Figure 13, cell c's value at time 7 is 30, the most recent value at time 5. A more conventional programming language view might be that this value should be considered to be undefined at times 6-8, but that is not the way time works in the real world. In the real world, slower moving objects don't have "undefined moments"; each intermediate state just lasts longer. This is visually depicted by the width of each value of slow cells, as in Figure 13.

Rule 2. What t means: For time indices in formulas (e.g., B<t-2>), t is an index on the global t-axis.

Rule 3. What constants mean: For time indices in formulas (e.g., B<t-2>), constants follow the units of the global t-axis.

Intuitively, t means "now". For example, if C refers to B<t-2>, then C's value at time 5 will be the same as B's value at time 3, as in Figure 13.

Rule 3 may seem puzzling: why is it necessary in a programming language to say what a constant means? The answer is that the multiple speeds introduce multiple unit possibilities. An alternative possibility for Rule 3 would be that constant 1 was 1 speed unit of the *referencing* cell; C's in the above example (slow, i.e. 1 constant unit = 4 ticks). Another possibility would be that constants were in the speed units of the *referenced* cell; B's in the above example (normal, i.e., 1 constant unit = 2 ticks). The advantage to the Rule 3 selected here is that it avoids programming language problems such as loss of referential transparency (e.g., a constant "3" meaning different things in different formulas), and that it explicitly relates to the t-axis indices visible in the VPL. We will also consider both other possibilities for Rule 3 later in this chapter.

4.2.2 Crab animation example

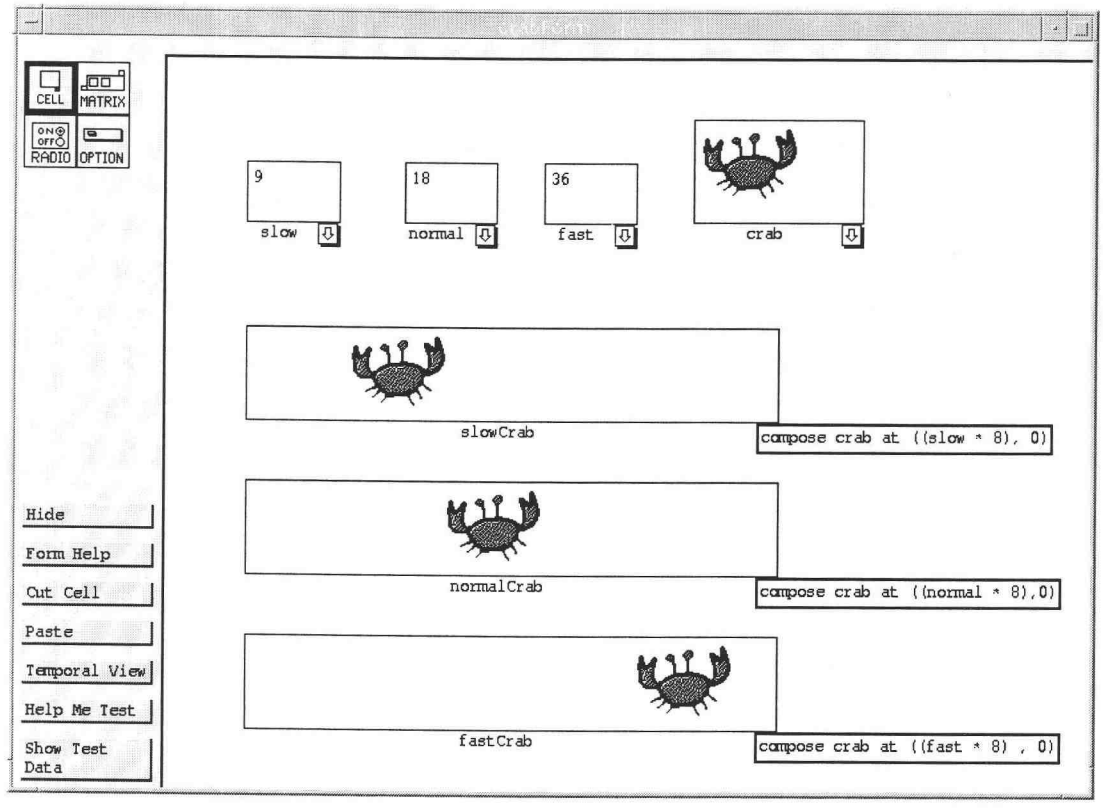


Figure 14: Crab animation example

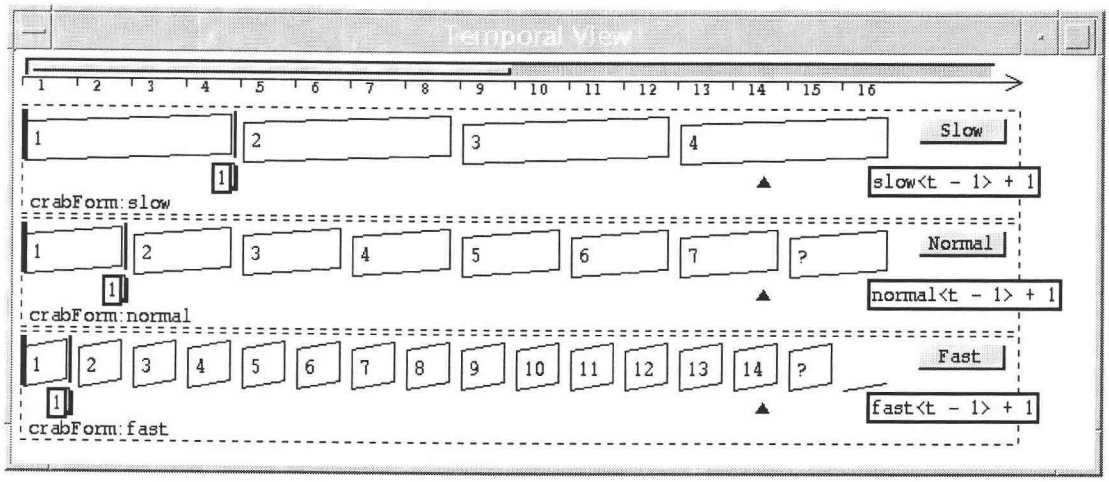


Figure 15: Temporal view of the three x-position cells in the crab animation example.

Figure 14 and Figure 15 show a simple animation under the SNF model. There are three x-position cells named `Slow`, `Normal`, and `Fast` in this form. They have similar formulas (increasing the previous element by 1 each step) but different speeds. Then the other three cells `slowCrab`, `NormalCrab` and `FastCrab` show the animated crabs moving to the right at different speeds by composing cell `crab` and referencing the x-position cells. From the temporal view in Figure 15, users can see the temporal vectors of the three x-position cells going at different speeds.

4.2.3 Evaluation

Under this set of rules, the SNF model adds the functionality needed to solve Problem 2, provided that only a fixed number of speeds are needed. (The noticeable improvement as a result will later be illustrated by Figure 18.) In this respect, the SNF model takes an important step forward from the base model. It retains the base model solution to Problem 1, but now also solves Problem 2 when only a fixed number of speeds are needed.

In Figure 12, Figure 13 and Figure 15, the width of the boxes depicts the relationships among different cells over time. Thus as in the base model, the visibility aspect of temporal relationships raised via introducing speeds is improved over "`fby`" and "`earlier`" notations.

However, the fixed number of speeds, which at first glance may seem only mildly limiting, is quite problematic from the standpoint of two of the cognitive dimensions, premature commitment and viscosity.

Premature commitment means users must make programming decisions before they have enough information to do so. In the SNF model, users only have limited choices of speeds, and they must decide upon the speed of cells (or accept the default decision of Normal speed) even if they do not yet know if they will eventually want to add faster or slower cells related to it. In the MatrixSort example, users might first choose Normal speed for the algorithm and Fast speed for the animation. But if another, even faster animation is also needed for the sort algorithm, users have no faster speed to choose. So they must change the sort algorithm to Slow speed and the original animation to Normal speed. Adding this new faster animation to the MatrixSort example introduces a lot of changes of cells' speeds, which demonstrates viscosity, or resistance to change.

	Problem 1	Problem 2	Visibility	Premature commitment	Viscosity
SNF model	Solved	Solved	Same as the base model	Unsolved	Unsolved

Table 2: Summary for the SNF model compared with the base model

Fortunately, removing the limitation on the number of speeds can help with these problems.

4.3 k*N model: k times faster/slower than Normal

The k*N model extends the SNF model such that any cell's speed can be specified as an arbitrary constant times faster or slower than Normal speed; hence it removes the limitation from the SNF model.

Unfortunately, this straightforward extension leads to a new problem: what to do about the integer numbering specified in Rule 1. If Normal speed means 1 value every c ticks, then whenever the user selects a k that is > c, non-integer indices arise. For example, if Normal speed means 1 value per tick and a cell `a` was specified to be 3 times faster than Normal, it would need 3 values per tick, leading to t-axis numbering like (1, 1.333, 1.666, 2, 2.333, ...).

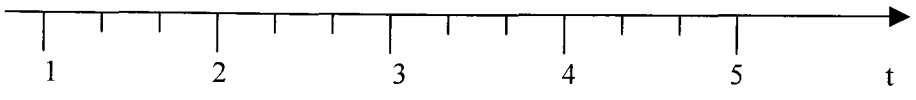


Figure 16: Only show integer numbers on the t-axis.

It might seem possible to keep only integer indices on the t-axis and not to show fractional numbers on it, like a ruler shown in Figure 16. However there still is a problem: users need fractional numbers in the formulas. For example, if the same cell `a` from the previous paragraph needs to reference its own previous element, the formula would be `<t-0.333>` since cell `a` is 3 times faster than the global speed. Here the consistency cognitive dimension resurfaces as an issue, since this numbering system is clearly not consistent with the usual numbering of x- and y-axes in grid-oriented programming languages.

One potential solution might be to eliminate "`faster than`" as an option, allowing only two speed choices: "`Normal`" or "`k * slower than Normal`". This retains integer numbering, but unfortunately also retains the premature commitment and viscosity problems of the SNF model, since now a commitment is needed that, if cell `a` is at Normal speed, no other cell will need to go faster than `a`.

A second potential solution is to retain integer numbering by automatically renumbering the t-axis indices when necessary. The problem here is that all the formulas relying upon a previous numbering would also have to be automatically changed. For example, consider cell `c`'s formula of `b<t-2>` in Figure 13. If the user then added a new cell `x` with "`4 times faster than Normal`" — not referencing or being referenced by `a`, `b`, or `c` — she would probably be surprised if the system automatically changed `C`'s formula to `b<t-8>`. Yet, making this change is necessary: if the system did not make this change, `c`'s formula

would produce different answers than it had before. Unfortunately though, even this is not enough: if another, unrelated spreadsheet, which was not automatically changed because it was not loaded at the time, was later loaded, its answers would now be wrong. This is a classic case of hidden dependencies, the cognitive dimension that identifies dependencies that are not explicit in the program. If any formulas' temporal indices include constants, these constants will depend on the global t-axis based on Rule 3 since all the formulas of these cells are tied to the global t-axis. Introducing a faster cell not only requires renumbering the global t-axis, but also all these formulas.

	Problem 1	Problem 2	Fractional numbers in formulas	Premature commitment	Viscosity	Hidden dependen-cies
k*N model	Solved	Solved	Unsolved	Improved over the SNF model	Improved over the SNF model	
The first potential solution of the k*N model (no "faster than")	Solved	Solved	Solved	Same as the SNF model	Same as the SNF model	
The second potential solution of the k*N model (renumber t-axis when needed)	Solved	Solved	Solved	Improved over the SNF model	Improved over the SNF model	Unsolved

Table 3: Summary for the k*N model with the first two potential solutions

4.4 Revising the k*N model via Rule 3

4.4.1 Description

Following the same strategy of the second potential solution (automatically renumbering the t-axis indices), it is possible to leave Rule 1 unchanged by instead changing Rule 3, what constants mean. Until now, constants have been in t-axis ticks; e.g., <2> meant the 2nd tick, and <t-2> meant 2 ticks before t. However, another look at Figure 13 and Figure 15 shows that there are multiple granularities along this axis. This presence of multiple granularities is significantly different from the way values populate x- and y- axes of grid-like systems. For example, in Figure 13, c's reference to b<t-2> at tick 9 on the t-axis is b's value 2 ***ticks*** back, rather than b's value 2 ***values*** back. Changing Rule 3 allows the user to work directly in the latter granularities:

Rule 3 (revised). What constants mean: For time indices in formulas (e.g., b<t-2>), constants are mapped onto the units of the global t-axis by multiplying by the granularity of the *referenced* cell.

For example, if b's granularity is 2 times slower than the t-axis, populating the t-axis at t=1,3,5... as in Figure 17, then c's reference to b<t-2> would, instead of as in Figure 13, refer to b 's temporal vector at t=1-(2*2), 5-(2*2), 9-(2*2), ... See Figure 17.

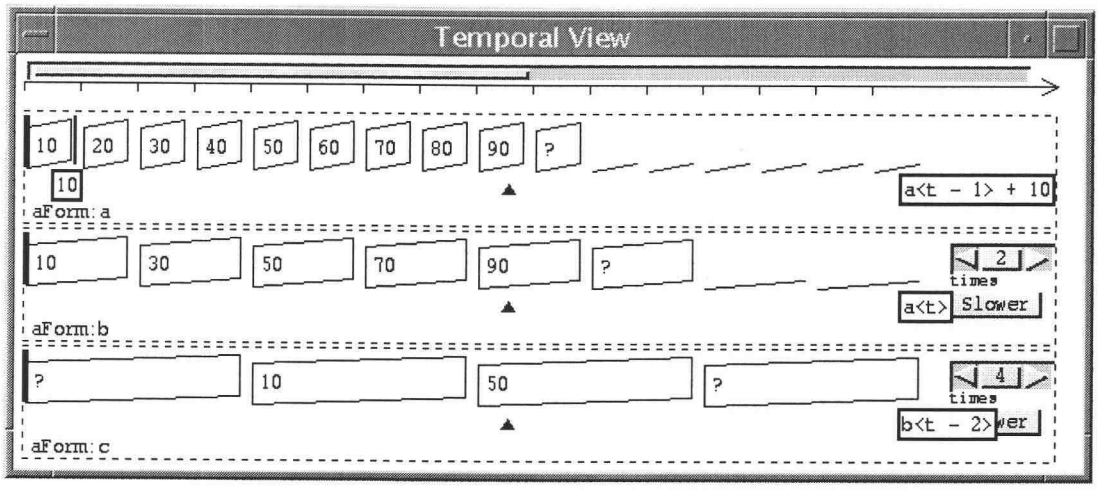


Figure 17: k*N model revision via Rule 3

In Figure 17, note that the t-axis numbering is not needed, since it no longer related to the meaning of the constants in the formulas due to this revision.

4.4.2 Evaluation

With this revision the t-axis can be renumbered whenever doing so is necessary to retain integer numbering of the t-axis, and renumbering will not require any formulas to be changed, which removes the hidden dependencies between the global axis and the formulas. So by this revision, the hidden dependencies cognitive dimension is improved over the original k*N model.

Premature commitment and viscosity are present to a lesser extent than in the SNF model. For example, unlike in the SNF model, in order to put in a faster animation within the MatrixSort example, the user only needs to specify the faster speed for the new animation. The remainder of the program remains unchanged. Note, however, that premature commitment is still an issue to some extent, because if two cells' speeds exist at adjacent integer factors (e.g., 2 times faster than Normal and 3 times faster than Normal), it is not possible to insert a new speed between those two, (e.g. 2.5 times faster than Normal) without redesigning all the speed

interrelationships in the existing program (a lot of work for this small change). Thus premature commitment and viscosity are not fully solved, but are improved.

This model has a compelling advantage: it avoids or reduces many of the difficulties of the previous potential solutions and solves both Problems 1 and 2. Figure 18 demonstrates this with the new version of the sort. Compare this figure to Figure 2: all the former `if`'s devoted to slowing down the timing are gone, and now only 3 formulas, all of which are non-nested, remain in the sorting program. One selects the `smallest` unsorted element, one removes it from the `unsorted` grid, and one appends it to the `sorted` grid. These grids' speeds are specified as Normal speed, and the animation cells' speeds are all specified as 10 times faster than Normal speed.

Figure 19 shows the temporal view of matrix `sorted` and the relevant subcell in matrix `sortOutput`. The second cell's speed is 10 times faster than the matrix `sorted`. Initially there is no value in matrix `sorted`, thus that subcell of matrix `sortOutput` has a static box value from time 1 to time 10. Start from time 11, when the value 3 comes to matrix sorted, the box begins to move to render the animation.

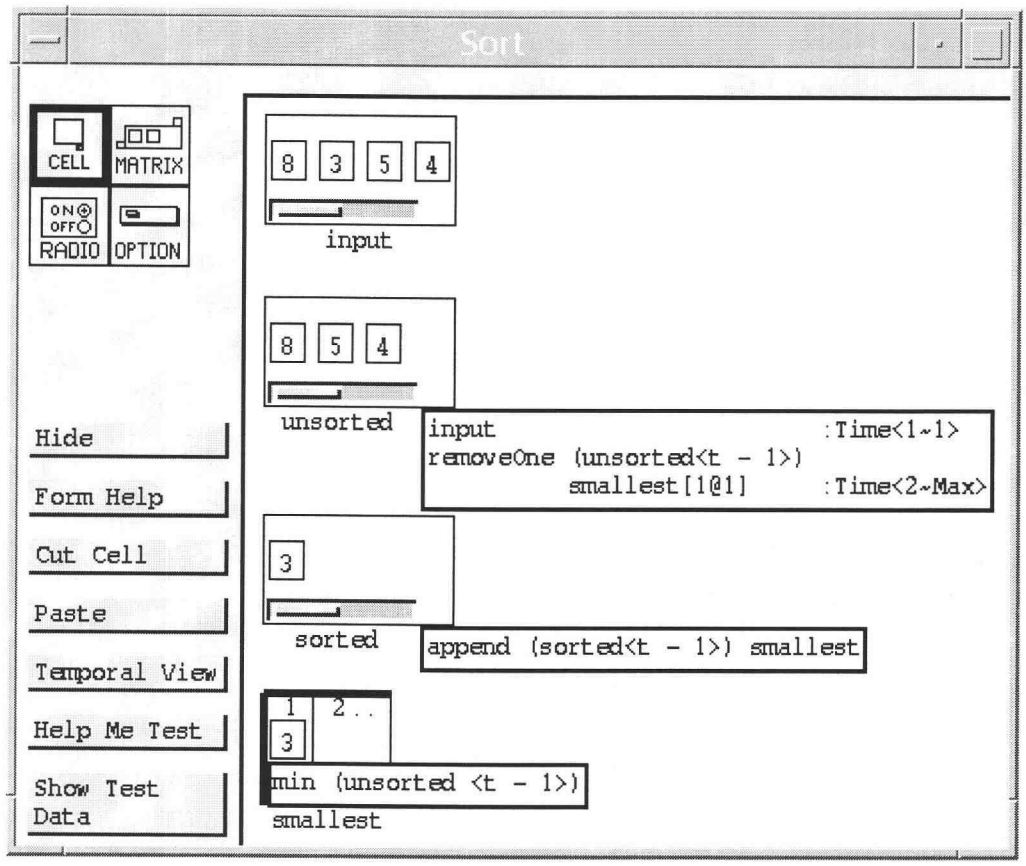


Figure 18: The selection sort under k*N model revision via Rule 3. See Figure 19 for a temporal view.

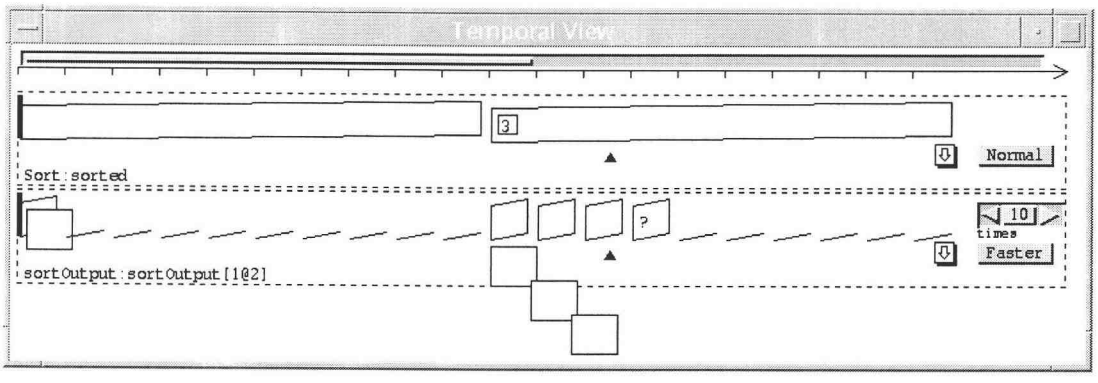


Figure 19: Temporal view of the sorted grid (top row) and the cell containing the box moving through different positions 10 times faster than normal (bottom row).

However, there are some other disadvantages. This variant of Rule 3 is less explicitly tied to the visible t-axis numbering. The loss of explicitness is because the subtraction of 2 from t in "<t-2>" is actually in different units than before: number of values rather than number of ticks.

The revision to Rule 3 also might seem to threaten referential transparency (the ability to substitute equals for equals), since the meaning of constants varies contextually. In Figure 17, if cell c's formula were "a<t - 2> + b<t - 2>", then the "a<t - 2>" value at time 9 would be cell a's value at time 7 (a<9- 2*1>=a<7>=70). The value for "b<t - 2>" would be cell b's value at time 5 (a<9- 2*2>=b<5>). Here "<t - 2>" means different referencing time positions based on referenced cells. But the time reference formulas (<t - 2>) are never isolated from the context (explicitly referenced cells). Since they always occur together, they always represent the same thing. (e.g. a<t - 2> always means cell a's second value before the current one). Thus referential transparency is not affected after all.

	Problem 1	Problem 2	Premature commitment	Viscosity	Hidden dependen-cies
k*N model revision via Rule 3	Solved	Solved	Improved over the SNF model	Improved over the SNF model	Solved

	Visibility	Referential transparency
k*N model revision via Rule 3	Worse than original the k*N model (hidden t-axis)	Unsolved for temporal subexpressions such as "<t-2>". But it is not an issue for entire expressions such as "a<t-2>".

Table 4: Summary for k*N model revision via Rule 3

4.5 Revising the k*N model via graphics-spatial programming

4.5.1 Description

By now it is clear that the t-axis differs from the x- and y-axes in having varying granularities in indexing. In grid-spatial programming, x- and y-axes do not have varying granularities. But, in graphics-spatial programming, the user can use different spatial granularities in different coordinate systems. That is similar to temporal programming at different speeds. So in this section, we will consider reapplying graphics-spatial programming techniques to temporal programming, working to make temporal programming consistent with graphics-spatial programming combined with grid-spatial programming. (Here we extend the research area to grid-based VPLs with graphics-spatial programming support, such as AgentSheets [16], Cocoa [13], Forms/3, NoPump [31], Spreadsheet for Information Visualization [6], and Spreadsheet for Images [19].) Graphics-spatial programming in Forms/3 was introduced in Section 3.3. Recall that every cell has its own coordinate system. By referencing other cells' graphics, a cell translates these graphics to its own coordinate system. The three classic basic graphics operations are translate, scale and rotate. (The graphics operations scale and rotate are not supported by Forms/3 yet, but only because there is no urgent need for them yet.) In this section we introduce similar operations into temporal programming in Forms/3.

Graphics-spatial programming in Forms/3 is done by the `compose` operator, which implements translate. For example, cell `b` can reference cell `a`'s graphics by "`compose a at (x,y)`". Here, x and y are the offsets in cell `b`'s coordinate system. Similarly, if the temporal vector is treated like a horizontal line composed of value points instead of pixel points, we can translate or scale it to another cell's t-axis. Here we add a new "`timeCompose`" operator with a t-offset parameter. If a user wants to reference cell `a`'s temporal vector, he or she can complete it by "`timeCompose a at my <temporal offset>`". (The "`at`" is replaced by "`at my`" in order to emphasize that the temporal offset is in the referencing cell's temporal scale). To implement the same example as in Figure 17, we have the formulas shown in Figure 20:

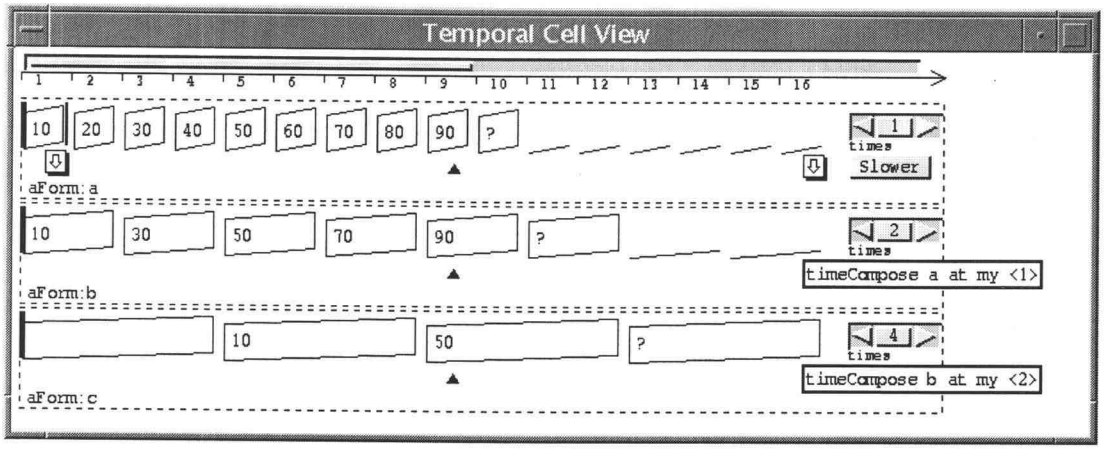


Figure 20: Translate in k*N model revision via graphics-spatial programming

In cell c's formula "timeCompose b at my <2>", the offset in the t-axis is in cell c's temporal coordinate system. This is consistent with graphics-spatial programming, in which the x- and y offsets after the "at" are in the referencing cell's own coordinate system. Thus, it is consistent here for "<2>" to mean translating cell b's temporal vector in cell c's temporal vector starting at the second value position of cell c.

When translating a picture from a high-resolution coordinate to a low-resolution coordinate, usually filtering will occur and some pixels are lost. Here for the same reason, in this translation from cell a to cell b, the values 20,40,60... are filtered out because of cell b's lower (temporal) resolution.

Another operation the user can do in graphics-spatial programming is scale. A new temporal operator "timeScale" performs the scale task for temporal coordinates. Scaling cell b's temporal vector to cell c's temporal vector is shown in Figure 21. With cell c's speed and cell b's speed already known, the system can automatically scale cell b's temporal vector to cell c's temporal vector. Unlike in the spatial scale operation, the temporal scale factor is known without the need for an explicit parameter.

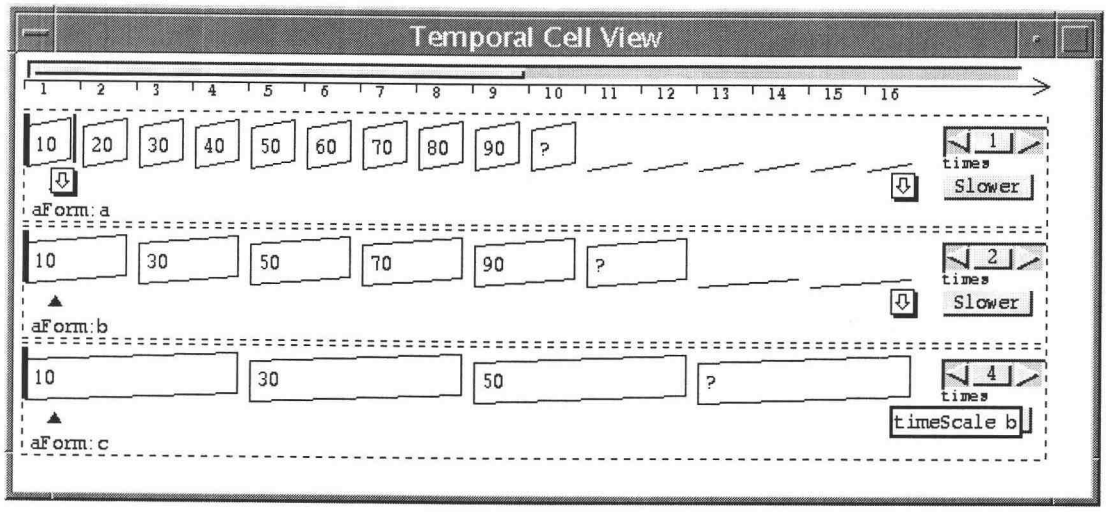


Figure 21: Scale in k*N model revision via graphics-spatial programming

Since time only has one dimension, it does not seem useful to introduce the rotate operation into temporal programming. Hence this third of the three classic graphics-spatial operations is omitted from our model.

4.5.2 Evaluation

Compared to the revision via Rule 3 of the k*N model, the most important advantage we gained in this revision is keeping consistency (with graphics-spatial programming instead of grid-spatial programming). Here the user can reapply graphics-spatial programming techniques to temporal programming. However, graphics-spatial programming is normally mastered by professional programmers rather than end users. Empirical work is needed to evaluate the usability of this revision of the k*N model for end users.

Regarding referential transparency, although there are no t offset notations in this revision, it remains an issue, because the same formulas can produce different results. For example, in Figure 22, cell `c`'s temporal vector is different with cell `b`'s temporal vector although they have the same formula "`timescale a`" but with different speeds. (Cell `b` is 2 times slower than Normal speed, cell `c` is 4 times slower than Normal speed.) So here referential transparency can be an issue for textual formulas alone, but not when the speeds are

combined with the textual formulas. This suggests that the speed specification should be considered part of the formula (at least in theory), which removes referential transparency as an issue.

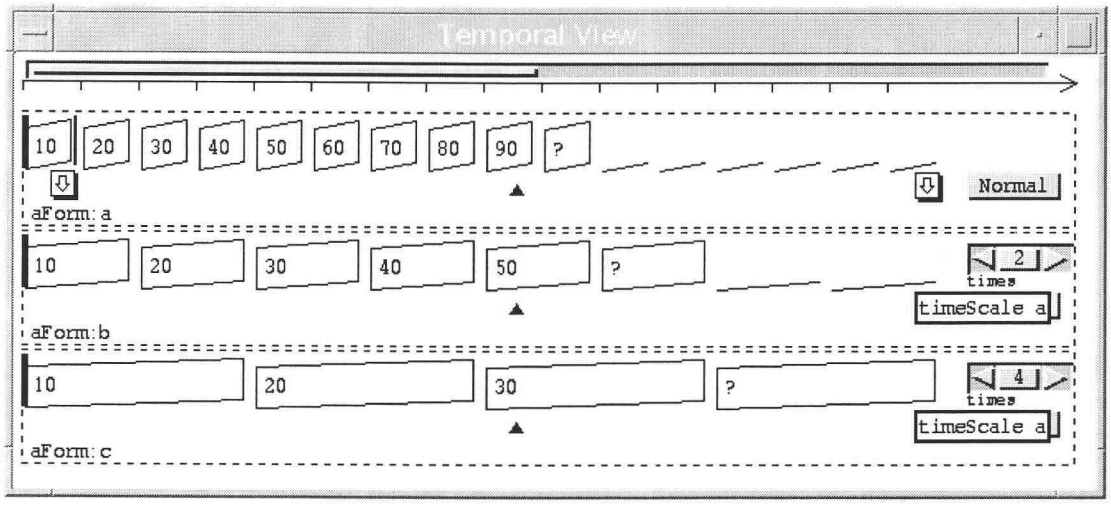


Figure 22: Referential transparency example in the k*N model revision via graphics-spatial programming

	Problem 1	Problem 2	Premature commitment	Viscosity	Hidden dependencies
k*N model revision via graphics-spatial programming	Solved	Solved	Improved over the SNF model	Improved over the SNF model	Solved

	Consistency	Referential transparency	High programming skills requirement
k*N model revision via graphics-spatial programming	Solved (consistent with graphics-spatial programming)	Unsolved for the textual formulas in isolation from speeds, but not an issue for the two together.	Unsolved

Table 5: Summary for k*N model revision via graphics-spatial programming

4.6 k*x model: k times faster/slower than x

4.6.1 Description

The k*x model is such a straightforward extension of the k*N model, few words are required to describe it; yet, it brings useful properties. The only difference from the k*N model is that, instead of specifying a cell's speed to be k times faster than Normal as in the k*N model, the user specifies the cell's speed to be k times faster than some other cell x. Thus, the spreadsheet of Figure 18 is exactly the same as that needed under this model. The temporal view of Figure 19 would be almost the same under this model, with the small change that, in the speed setting for the animation cell, the user would specify that it was 10 times faster *than* `sorted`.

The two revisions of the k*N model can be applied to this model too. Revising the k*x model via Rule 3 or revising the k*x model by introducing graphics-spatial like operations bring the same benefits and problems to the k*x model as to the k*N model.

4.6.2 Evaluation

The k*x model features better closeness of mapping than the previous model, because the user can specify the temporal relationships among cells directly, instead of translating them all into temporal relationships with Normal speed. Further, this model removes the problems of premature commitment and viscosity while retaining the advantages of solving Problems 1 and 2. There is no way to set a cell's speed at 2.5 times faster than Normal speed in the k*N model, which cause the premature commitment and viscosity problems. But in this model, assume cell `a`'s speed is 2 times slower than Normal speed, cell `b`'s speed is 5 times faster than cell `a`, then we can set up cell `b` to be 2.5 times faster than Normal speed.

However, with the improvement of closeness of mapping, the problem in hidden dependencies emerges to some extent. In previous time models, all cells' speeds are dependent on Normal speed. The user can know how fast a cell goes by looking at its speed selection on the right of the temporal view. However, in this time model, dependencies on other cells' speeds have replaced dependencies on Normal speed. The dependencies "Whose speeds are my speed dependent on?" are explicit, but the dependencies "Whose speeds are dependent on me?" are hidden. When a user needs to change one cell's speed, he or she does not know what other cells' speeds dependant on this cell's speed will be changed too.

	Problem 1	Problem 2	Closeness of mapping	Premature commitment	Viscosity	Hidden Dependencies
k*x model	Solved	Solved	Improved over the k*N model	Improved over the k*N model	Improved over the k*N model	Worse than the k*N model

Table 6: Summary for the k*x model

Chapter 5: Implementation Issues

This thesis introduces a continuum of time models to Forms/3. The first three have been implemented so far. Before this thesis, Forms/3 only supported one-dimensional ways to program in time with "fby" and "earlier" notations.

Each model's implementation is based on the previous model. With the temporal view implemented, Forms/3 supports the base model. In order to support the SNF model, new code was embedded in the temporal view and a notion of a "TVinterval" was introduced to the evaluation engine to represent speed. The k*N model is supported by further changes in temporal view and support for dynamically rearranging the global internal t-axis when the system needs a new faster internal speed. Both the revision via Rule 3 and the revision via graphics-spatial programming of this model are supported by changes in formula parsing and in the evaluation engine. The k*x model is not implemented.

Files temporalViewObj.lisp, temporalRegionObj.lisp, temporalComponentObj.lisp, and temporalFormulaWindow.lisp include all temporal view (GUI side) code. But other parts of the implementation of this thesis are located in many places. To help future students in the Forms/3 group to implement the k*x model based on previous work, this chapter explains what changes this thesis has made and what needs to be implemented in the future. This chapter is also useful for the students who want to understand the time models and evaluation engine in Forms/3. This chapter describes the implementation structure via sequence diagrams and concrete examples.

5.1 Base model

Integer time was already supported in the Forms/3 implementation before this thesis. The new things in the base model are how to support the temporal view and how to replace the "fby" and "earlier" notations with the new temporal formula notation. The user not only can see the temporal vector of the cell, but also can enter the formula to the cell using time

regions. The first subsection discusses the structure of the temporal view, and the next subsection explains how it works for users.

5.1.1 Structure of the temporal view

The temporal view is composed of a temporalFormulaWindow, temporalViewObj, temporalRegionObj and temporalComponentObj. See Figure 23:

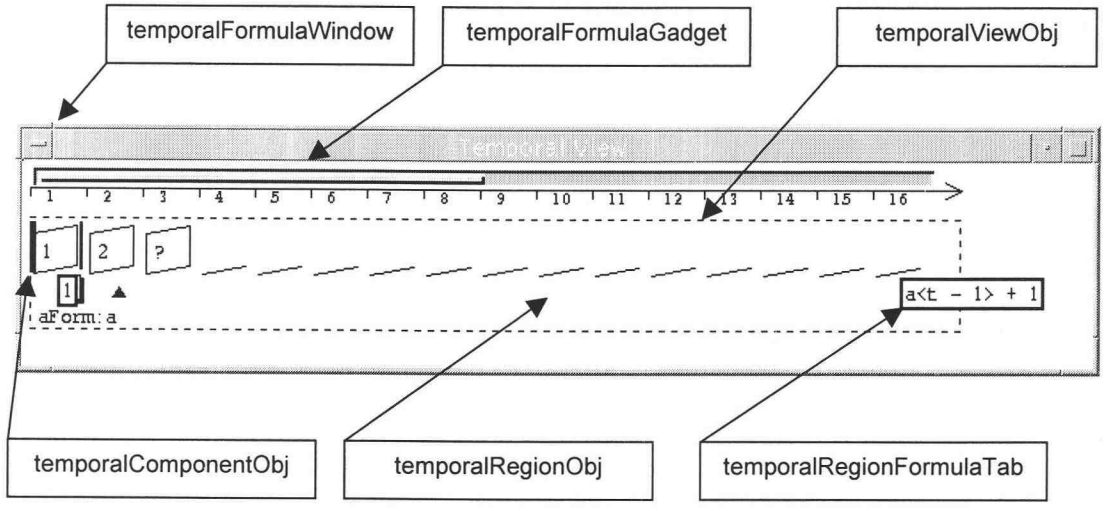


Figure 23: The positions of temporal objects

In Garnet data structures (Garnet is one of the current GUI developing environments of Forms/3), the "has a" relations among them can be described as in Figure 24:

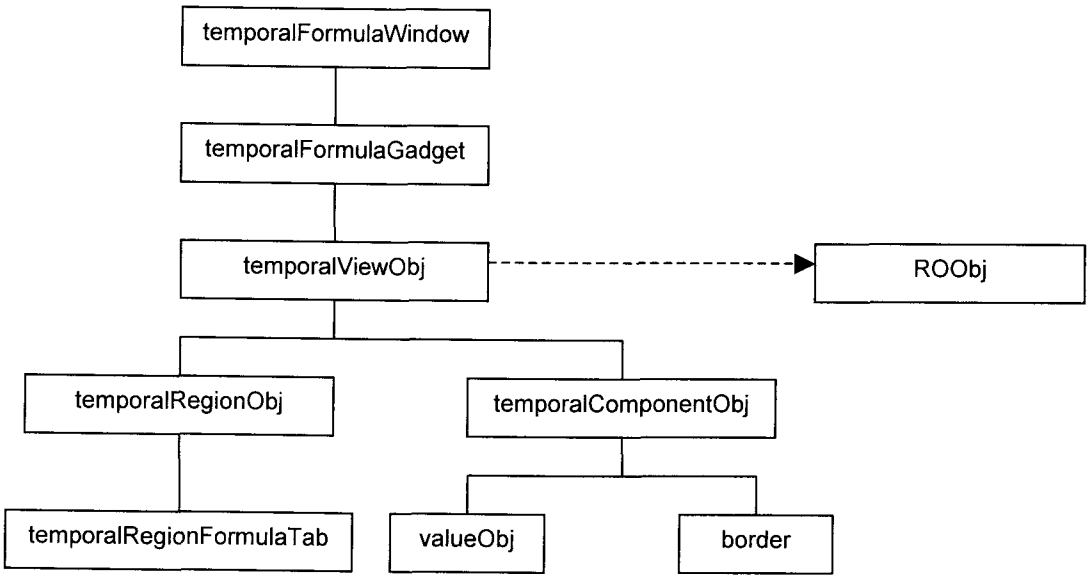


Figure 24: Structure of temporal view

The temporalFormulaWindow represents the window of the temporal view. The temporalFormulaGadget is the container of the temporalViewObjs, which represent the temporal vector of one cell in the temporal view. The temporalViewObj is the container of both the temporalRegionObj (which represents a region and holds a region formula tab) and the temporalComponentObj (which represents the parallel rectangle and holds the value object displayed in it). Another choice might have been that the temporalRegionObjs were contained in the temporalViewObj and the temporalComponentObjs were contained in the temporalRegionObj. However, when users create a region or delete a region, rearranging the containing relationships between temporal components and temporal regions would have been much more difficult than the first choice. Instead, the system only needs to create or delete temporalRegionObjs, and the temporalComponentObjs need not to be concerned.

There is a pointer from the temporalViewObj to the ROObj. That is the only way the Temporal View structure knows about the data structure in the engine side. It is not strongly tied to the engine side in order to make it easier for the JavaForms implementation to support the temporal view in the future. JavaForms needs the GUI code to be easily separated with as little as possible intermingling between the GUI and the engine.

5.1.2 What happens when users specify a formula?

In Figure 25, the sequence diagram shows the action sequence (before the implementation in this thesis) when the user specifies a formula. To support specifying temporal formulas in the temporal view, changes are added in four places, marked in Figure 25.

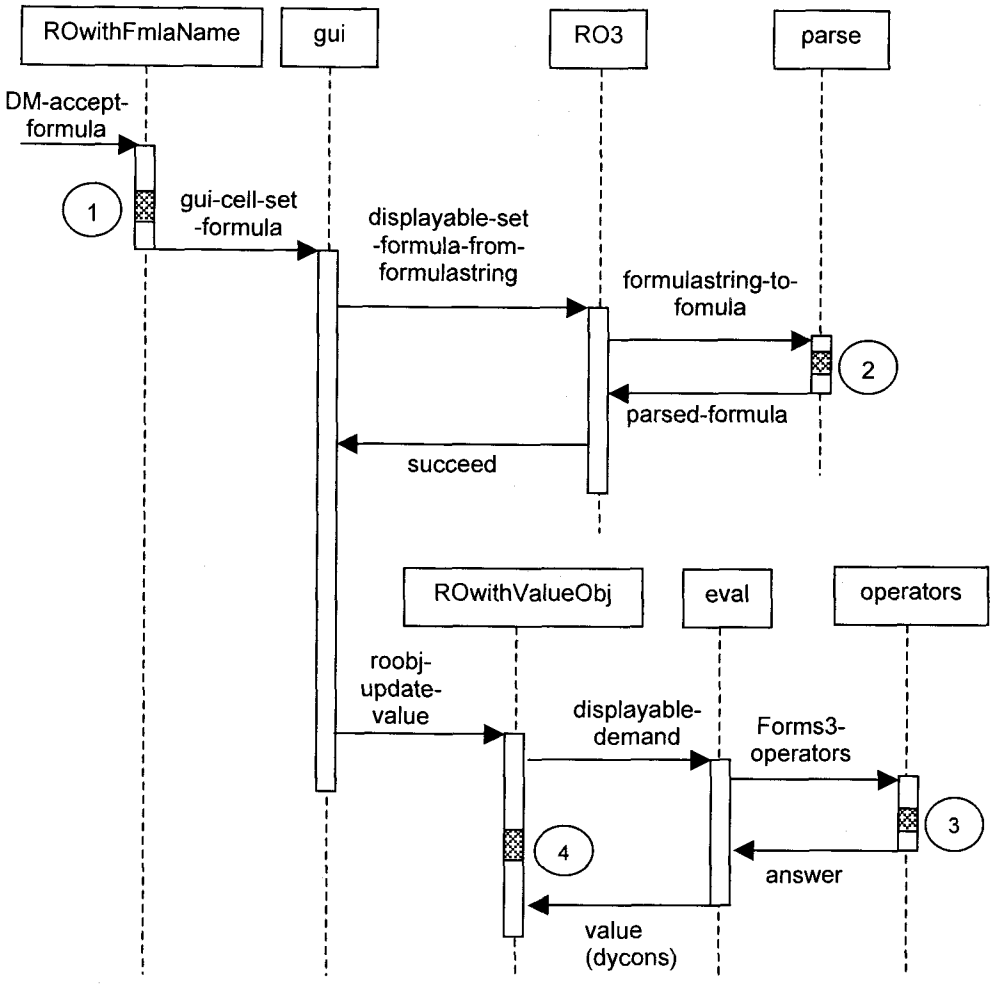


Figure 25: Sequence diagram when the user specifies a formula. The numbers mark the places changes were needed.

Change mark 1. Generate temporal formula based on temporal region indices:

A concrete example is described here to explain the sequence diagram in Figure 25. Assume cell a's formula is specified by the user as shown in Figure 26. The initial value of cell a is 1. After that, every time tick, a's value will be increased by 1.

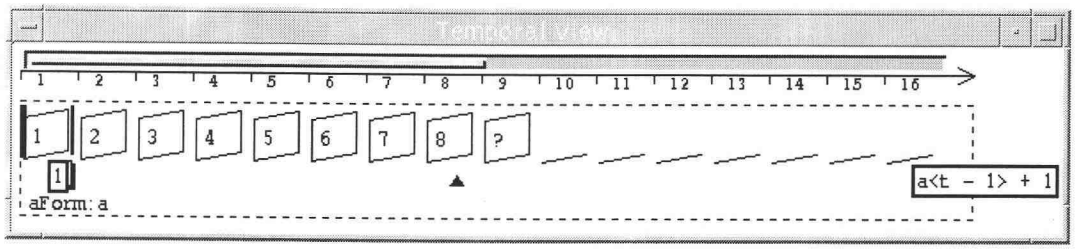


Figure 26: Cell a in the base model

When the user accepts the formula shown in Figure 26, DM-accept-formula is called. As soon as the system knows the user is editing a formula in the temporal view, it calls temporalViewObj-set-formula (in temporalViewObj.lisp). The temporalViewObj-set-formula calls temporalViewObj-gather-formulaString which combines the temporal region formulas with the temporal region indices: "`1 :Time{1~1} #\Newline a<t - 1> + 1 :Time{2~Max}`". The temporalViewObj-set-formula calls gui-cell-set-formula with that formula. The sequence is described in Figure 27.

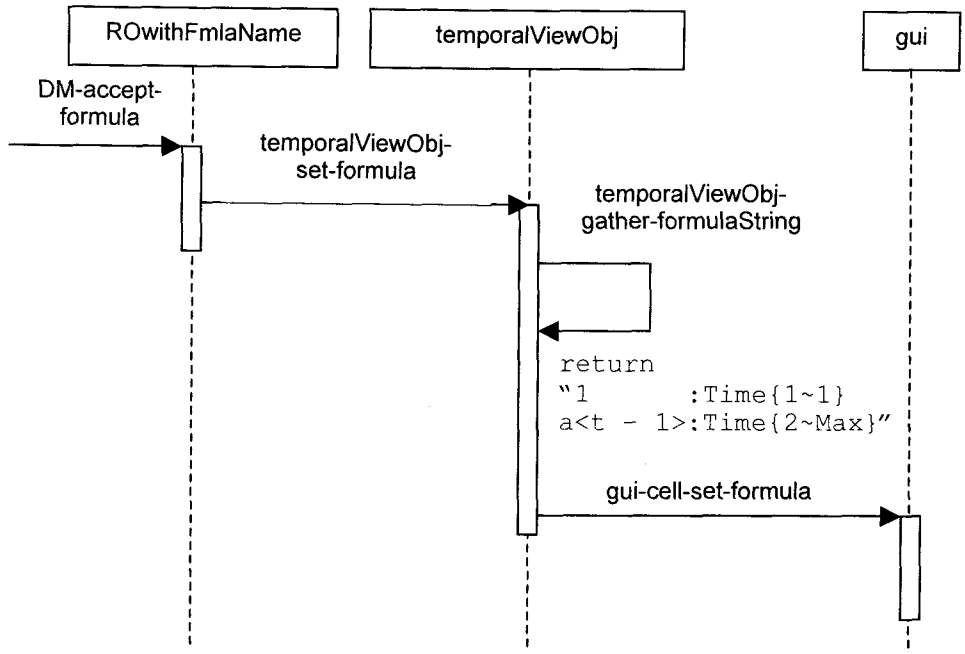


Figure 27: Sequence diagram of change mark 1

Change mark 2. Special formula parsing code for temporal formulas:

In parse.lisp, new parsing code for this format of formulas is needed. After parsing, the internal formula of cell `a` is: "`(TemporalRegion ((1) 1 1) (((+ (TimeRef #S(CellRef Form NIL cellID CELL31669343-5362) (-(SeeTime) 1)) 1)) 2 999999))`".

Change mark 3. New temporal operators FormsTimeRegion and FormsTimeRef:

In Figure 26, cell `a`'s value is demanded at time (2). As the internal formula alone suggests, two new temporal operators have been implemented in operators3.lisp. FormsTimeRegion (corresponding to TemporalRegion) selects the appropriate temporal region to compute the cell's value at demand time. In this example, the second temporal region's formula "`a<t - 1> + 1`" is the appropriate one, and it demands "`a<t - 1>`" at time 2. Then FormsTimeRef (corresponding to TimeRef) is called to evaluate it, which demands cell `a`'s value at the time being demanded minus 1 (`2-1=1`). FormsTimeRegion is called again, and this time the first temporal region formula "`1`" is picked so the answer is 1. With that returned answer from FormsTimeRef, the first call to FormsTimeRegion returns 2

(a<t-1>+1=1+1=2), Displayable-demand returns a dycon with value 2 in it. (A dycon is a value object that knows how to paint itself on the screen.)

Change mark 4. Update the temporal view:

In roobj-update-value, if the cell is shown in the temporal view, the temporalViewObj-update is called to update it. Thus the value 2 is displayed to both the cell on the form and the temporal vector in the Temporal View.

5.2 SNF model

In the SNF model, in order to support the Slow, Normal and Fast speeds in Forms/3, a notion of TVinterval was added to each RO. It represents the interval of the RO's temporal vector under the global t-axis. If an RO is set to Fast speed, its TVinterval is 1; if Normal speed, its TVinterval is 2; if Slow speed, its TVinterval is 4. In the temporal view, a radio button with Slow, Normal and Fast options was added (shown in Figure 12).

Fast speed allows values at every tick, Normal speed allows values at every 2 ticks, (1,3,5...), and Slow speed allows values at every 4 ticks (1,5,9...). Two places call translate-time-by-interval to ensure that the define time and expire time fall on the correct ticks. (e.g. Cells with Slow speed or Normal speed cannot have define time or expire time at time tick 2.) One is in eval3.lisp displayable-compute to translate the expire time. The other one is in operator3.lisp FormsTimeRegion to translate the define time.

When the user selects the speed for a cell, the sequence is as shown in Figure 28.

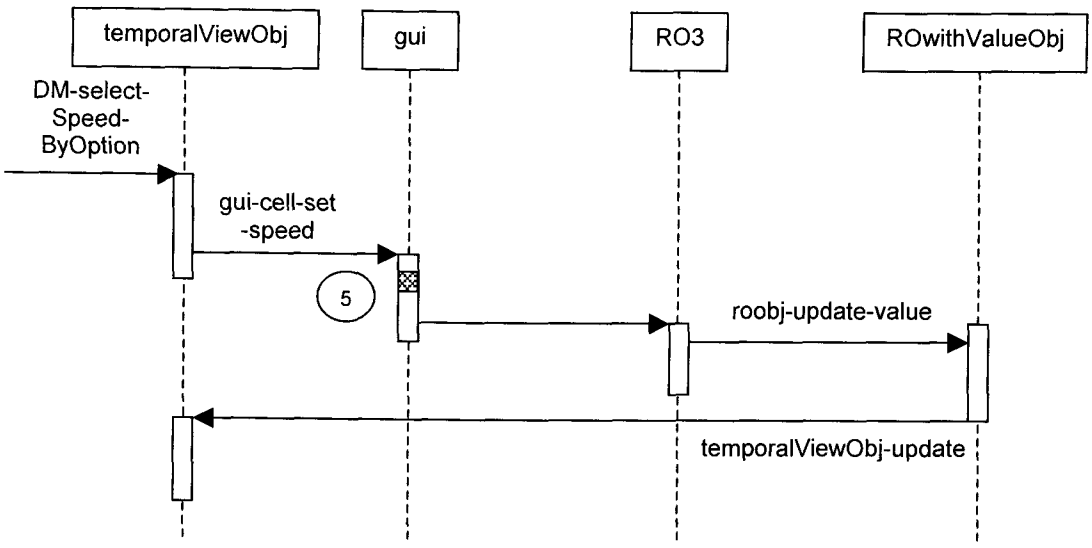


Figure 28: Sequence diagram when the user specifies the cell's speed. The number 5 marks the change needed in the k*N model, which is discussed in the next section.

First, the DM-select-Speed-ByOption notices the speed selection (Slow, Normal and Fast) by the user. Then it calculates the interval and passes it to gui-set-cell-speed. Gui-set-cell-speed changes the TVinterval in the RO object, destroys all cells' temporal vectors that are dependent on this cell, then calls displayable-force-gui-update. Displayable-force-gui-update collects all affected cells from the WAW table and calls displayable-demand-on-screen. Only for the on-screen cells, roobj-update-value is called to update the cells, which finally calls temporalViewObj-update to update the cells in the temporal view.

5.3 k*N model

The original k*N model was not implemented because of the fractional number t-axis problem. Instead, two revisions of this model are implemented.

The revision via Rule 3 specifies that the constant in the time indices maps onto the units of the global t-axis by multiplying by the granularity of the *referenced* cell. The necessary changes were made in the FormsTimeRef operator in operators3.lisp. When this operator calculates the referenced time position, it multiplies the offset constant by the interval of the referenced cell.

In this revision of the k*N model, when the user introduces a faster speed, dynamically rearranging the global t-axis is necessary to avoid the fractional number t-axis problem described in Section 4.3. The new code is inserted at the position in Figure 28 marked with "dynamically rearrange t-axis in the k*N model". In gui-cell-set-speed, if a new cell's speed is faster than the global internal speed, it picks a new global interval speed, which is fast enough to represent all cells. Then dynamic-scale-global-t-axis (in time3.lisp) is called to rearrange the global internal speed.

After the system changes the TVinterval of all the cells, it destroys their temporal vectors, whose time cost is O(number of cells), because the implementation environment (lisp) has garbage collection. Garbage collection is not free of cost, but we treat it as if it were, because it is done during periods of inactivity, such as when the user pauses to think.

Then the system demands the current values of only producers of the on-screen cells. So the number of total demands is O(number of producers of onscreen cells). Because the costs of all the computational operators in Forms/3 are O(1) and there are a limited number of operators and operands in a formula with constant-bounded length, the cost of computation of one cell is O(1).

In the worst case, each producer will require its entire history to be recomputed. This is the same as the number of computations required under eager evaluation if all cells in the program affect on-screen cells. Since eager evaluation has less overhead, it would be more efficient in this worst case. However, eager evaluation's best case matches its worst case, whereas lazy evaluation's best case can be much less, since computations not required to keep the on-screen cells up-to-date are omitted.

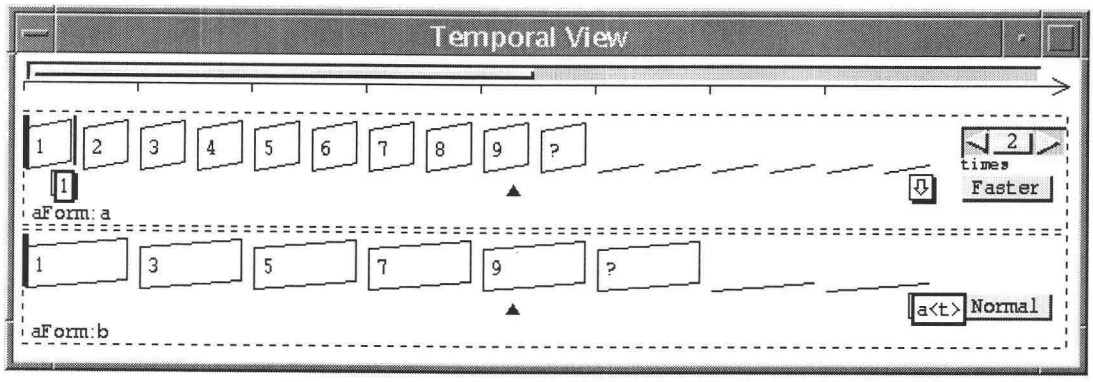


Figure 29: After the global t-axis rearranging

For example, assume the current global interval speed is the same as Normal speed when the user specifies cell a to be 2 times faster than Normal speed (shown in Figure 29). dynamic-scale-global-t-axis picks 2 times faster than Normal speed as the global interval speed. Then it multiplies all cells' TVinterval by 2, destroys their temporal vectors and demands the onscreen cells to update them. Since the constants in the time indices are relative to the referenced cell's TVinterval, this rearrangement of the interval global t-axis will not require changing the cell's formulas. This rearrangement is also needed in the revision via graphics-spatial programming and the k*x model.

The revision via graphics-spatial programming introduced new operators FormsTimeScale and FormsTimeCompose in operators3.lisp and the corresponding parsing code in parse.lisp.

5.4 k*x model

This model is not implemented at this time. What we need to do is introduce a new kind of dependency in the WAW-table – a temporal speed dependency. Then when a cell's speed is changed, displayable-force-gui-update will collect the cells affected by this speed change from the WAW-table and update them. Also two data members will be added to an RO, TVspeed-reference-cell and TVspeed-reference-times. For example, if cell a is 2 times faster than cell b, then TVspeed-reference-cell is cell b, and TVspeed-reference-times is 2.

This is needed because if cell a's temporal vector is destroyed, cell a's TVinterval needs to be recalculated, which can be redone by demanding cell b 's speed. If cell b's speed depends on other cells' speeds, this calculation needs to continue until the cell's TV-speed-reference-cell is null.

Chapter 6: Conclusions and Future Work

Although several prior researchers have made the point that temporal programming should be achievable using exactly the same mechanisms as spatial programming, this possibility has not been thoroughly investigated, because prior approaches all stay in global integer time scale with only one global speed. In this thesis, we have added to what is known about this possibility for grid-based VPLs by attempting to solve two specific problems in this way without interfering with cognitive aspects of programming. The motivation is this: end users succeed at grid-based programming in space, which suggests that an approach consistent with this for programming temporal (animation) behaviors would allow this audience to reapply what they already know to this new domain.

The models we have developed reveal several core differences between the ways grid-based programming have been supported in space and the extension of such ways to temporal programming:

- The ability to access *any* position on the x- or y-axis loses its consistency with the real world on the t-axis: what does it mean for a past position to reference the future?
- In the real world, in space, if a value is not present at some specific position then it is missing, whereas in time, a situation at time t persists until something different occurs. Upholding these real-world conventions in a VPL seems necessary and useful but introduces inconsistency between space and time.
- Whereas spatial grid element sizes are constant in the grids found in the VPLs of which we are aware, multiple granularities on the t-axis are needed to support varying speeds.

The third difference seems to be the most problematic. The first two differences did not lead to problems in the first model (base model), but because that model does not accommodate the third difference, it does not solve Problem 2 (Support program temporal interrelationships). Accommodating the third difference in the next two models introduced several problems such as premature commitment, viscosity, lost of consistency, lost of visibility, lost of referential transparency, high programming skills requirement and hidden

dependencies, all of which have been shown to cause difficulties for humans. (See the summary in Table 7 in Appendix A). The last model (the k*x model) with revision via Rule 3 supports direct, grid-based specification of an unlimited number of speeds and temporal relationships, solving both Problems 1 and 2. This revision of the model thus provides a measurable step forward in functionality for grid-based languages, albeit at a cost of lower visibility with spatial grids than that of the base model, necessitated by the granularity difference.

Our assumption is the revision via Rule 3 with the k*x model is more suitable for end users, and the revision via graphics-spatial programming is more suitable for professional users. Further empirical study is needed to prove or disprove this assumption.

Note that the models presented are all static in the sense that the cell's temporal speed can be statically determined, without the need for run-time data. We are also considering experimenting with an inherently dynamic model, a y*x model, in which a user can specify a cell's speed as an arbitrary computation `y` times faster/slower than the speed of another cell `x` (e.g., `cellA` goes `cellB` times faster than `cellC`).

Bibliography

[1] A. Ambler and A. Broman, Formulate solution to the Visual Programming Challenge, *J. Visual Lang. Computing* 9(2), 171-209, Apr. 1998.

[2] J. Atwood, M. Burnett, R. Walpole, E. Wilcox, and S. Yang, Steering programs via time travel, *IEEE Symp. Visual Lang.*, Boulder, CO, 4-11, Sept. 1996.

[3] M. Burnett, J. Atwood, Z. Welch, Implementing level-4 liveness in declarative visual programming languages, *IEEE Symp. Visual Lang.*, Halifax, Canada, Sept. 1998.

[4] M. Burnett and H. Gottfried, Graphical Definitions: Expanding Spreadsheet Languages through Direct Manipulation and Gestures, *ACM Transactions on Computer-Human Interaction* 5(1), 1-33, March 1998.

[5] P. Carlson, M. Burnett, and J. Cadiz, A seamless integration of algorithm animation into a visual programming language, *ACM Proc. Workshop Advanced Visual Interfaces*, Gubbio, Italy, 194-202, May 1996.

[6] E. Chi, J. Riedl, P. Barry, and J. Konstan, Principles for information visualization spreadsheets, *IEEE Computer Graphics and Applications*, July/Aug. 1998.

[7] W. Du and W. Wadge, A 3d spreadsheet based on intensional logic, *IEEE Software*, 78-89, May 1990.

[8] M. Duecker, C. Geiger, R. Hunstock, G. Lehrenfeld, W. Mueller, Visual-textual prototyping of 4d scenes, *IEEE Symp. Visual Lang.*, Capri, Italy, 328-335, Sept. 1997.

[9] R. Duisberg, Animated graphical interfaces using temporal constraints, *ACM Conf. Human Factors in Computer Systems (CHI'86)*, Boston, MA, 131-136, Apr. 1986.

[10] R. Duisberg, Visual programming of program visualizations: a gestural interface for animating algorithms, in *Visual Languages and Applications* (T. Ichikawa, E. Jungert, R. Korfhage, eds.), Plenum Publishing, NY, 1990.

[11] E. Freeman, D. Gelernter, and S. Jagannathan, Uniformity of environment and computation in MAP, *IEEE Symp. Visual Lang.*, Boulder, CO, 130-137, Sept. 1996.

[12] T. Green, M. Petre, Usability analysis of visual programming environments: a 'cognitive dimensions' framework, *J. Visual Lang. Computing*, 131-174, June 1996.

[13] N. Heger, A. Cypher, and D. Smith, Cocoa at the Visual Programming Challenge 1997, *J. Visual Lang. Computing* 9(2), 151-169, Apr. 1998.

[14] S. Hibino and E. Rundersteiner, User interface evaluation of a direct manipulation temporal visual query language, *ACM Int'l. Multimedia Conf.*, Seattle, WA, 99-107, Nov. 1997.

[15] E. Hutchins, J. Hollan, and D. Norman, "Direct Manipulation Interfaces," in User Centered System Design: New Perspectives on Human-Computer Interaction (D. Norman, S. Draper, eds.), Lawrence Erlbaum Assoc., Hillsdale, NJ, 87-124, 1986.

[16] A. Ioannidou and A. Repenning, End-user programmable simulations, *Dr. Dobb's Journal*, 40-48, Aug. 1999.

[17] K. Kahn, ToonTalk—An Animated Programming Environment for Children, Journal of Visual Languages and Computing 7(2), 197-218, June 1996.

[18] G. Kutty, L. Dillon, L. Moser, P. Melliar-Smith, and Y. Ramakrishna, Visual tools for temporal reasoning, *IEEE Symp. Visual Lang.*, Bergen, Norway, 152-159, Aug 1993.

[19] M. Levoy, Spreadsheet for images, *ACM Siggraph 94*, 139-146, 1994.

[20] R. MacNeil, Generating multimedia presentations automatically using TYRO, the constraint, case-based designer's apprentice, *IEEE Workshop Visual Lang.*, Kobe, Japan, 74-79, Oct. 1991.

[21] T. McCartney, K. Goldman, and D. Staff, EUPHORIA: End-User Construction of Direct Manipulation User Interfaces for Distributed Applications, *Software Concepts and Tools* 16(4), 1995, 147-159.

[22] R. McDaniel and B. Myers, Getting more out of programming-by-demonstration, *ACM Conf. Human Factors in Computing Systems*, Pittsburgh, 442-449, May 1999.

[23] R. Pandey and M. Burnett, Is it easier to write matrix manipulation programs visually or textually? An empirical study, *IEEE Symp. Visual Lang.*, Bergen, Norway, 344-351, Aug. 1993.

[24] J. Pfeiffer Jr., Altaira: A Rule-based Visual Language for Small Mobile Robots. *Journal of Visual Languages and Computing* 9(2), April 1998, 127-150.

[25] T. Smedley, P. Cox, and S. Byrne, Expanding the Utility of Spreadsheets Through the Integration of Visual Programming and User Interface Objects, *ACM Proceedings of the Workshop on Advanced Visual Interfaces*, Gubbio, Italy, 148-155, May 27-29, 1996.

[26] J. Song, M. Kim, and G. Ramalingam, Interactive authoring of multimedia documents, *IEEE Symp. Visual Lang.*, Boulder, CO, Sept. 1996, 276-283.

[27] N. Stankovic and K. Zhang, Towards visual development of message-passing programs, *IEEE Symp. Visual Lang.*, Capri, Italy, 144-151, Sept. 1997.

[28] G. Viehstaedt and A. Ambler, Visual representation and manipulation of matrices, *J. Visual Lang. Computing* 3(3), 273-298, Sept. 1992.

[29] W. Wadge and E. Ashcroft, *Lucid, the Dataflow Programming Language*, Academic Press, London, 1985.

[30] G. Wang and A. Ambler, Solving display-based problems, *IEEE Symp. Visual Lang.*, Boulder, CO, 122-129, Sept. 1996.

[31] N. Wilde and C. Lewis, Spreadsheet-based interactive graphics: from prototype to tool, *ACM Conf. Human Factors in Computing Systems*, 153-159, Apr. 1990.

[32] D. Wolber, Pavlov: an interface builder for designing animated interfaces, *ACM Trans. Computer-Human Interaction* 4(4), 347-386, Dec. 1997.

Appendices

Appendix A – Summary table for all time models

	Problem 1	Problem 2	Closeness of mapping	Consistency	Visibility	Premature commitment
Base model	Solved	*Unsolved*	Improved over Forms/3 prior to this thesis	Improved over Forms/3 prior to this thesis	Improved over Forms/3 prior to this thesis	Does not cause this problem
SNF model	Solved	Solved				*Unsolved*
k*N model	Solved	Solved				Improved over the SNF model
k*N model revision via Rule 3	Solved	Solved			Worse than the SNF model	
k*N model revision via graphics-spatial program-ming	Solved	Solved		Consistent with graphics-spatial program-ming		
k*x model	Solved	Solved	Improved over the k*N model	See both the k*N model revisions	See both the k*N model revisions	Improved over the k*N model

Table 7: Summary for all time models. Blank means approximately the same as the above row.

	Viscosity	Fractional numbers in formulas	Hidden dependency	Loss of referential transparency	High programming skills background
Base model	Does not cause this problem	Does not cause this problem	Does not cause this problem	Does not cause this problem	Does not cause this problem
SNF model	*Unsolved*				
k*N model	Improved over the SNF model	*Unsolved*			
k*N model revision via Rule 3		Solved	Solved	*Unsolved* But only for temporal subexpressions	
k*N model revision via graphics-spatial programming				*Unsolved* but only for the textual formulas isolated from speeds	*Unsolved*
k*x model	Improved over the k*N model		Worse than the k*N model	See both the k*N model revisions	See both the k*N model revisions

Table 7 (Continued): Summary for the all time models

Appendix B – Run Forms/3 with different time models

1. Run Forms/3 under the base model

The default option is run with the base model.

run GarnetForms

(load "RUN")

(run)

2. Run Forms/3 under the SNF model

Run Forms/3 with the :speedModel set to be :SNFModel.

run GarnetForms

(load "RUN")

(run :speedModel :SNFModel)

3. Run Forms/3 under the k*N model with revision via rule3

run GarnetForms

(load "RUN")

(run :speedModel :k*Nmodel :speedModelRevision :revision1)

4. Run Forms/3 under the k*N model with revision via graphics-spatial programming

run GarnetForms

(load "RUN")

(run :speedModel :k*Nmodel)